MASTER EN SCIENCES INFORMATIQUES

LonOS: un système d'exploitation orientée-objets de points de contrôle distribués

Quoitin, Bruno

Award date:
1999

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

LonOS

Un système d'exploitation orientée-objets de points de contrôle distribués

Bruno QUOITIN

Version du 29 mai 1999

Travail de fin d'études réalisé en vue de l'obtention
du titre de Maître en Informatique

Année Académique 1998-1999

Résumé

Le système LONOS est une couche logicielle destinée à faciliter l'exploitation des réseaux de points de contrôle par des applications (clients) et ceci avec une approche orientée-objets. LONOS permet aussi la liaison de ces réseaux de points de contrôle avec des réseaux locaux voire Internet.

L'exploitation orientée-objets de ces réseaux de points de contrôle permet de prendre en compte les fonctionnalités très diverses des ressources (équipements de contrôle) connectées à ces réseaux. Ces différences de fonctionnalités sont dues à la provenance et aux usages variés de ces noeuds (éclairage, alarme, etc). Ces ressources sont représentées par des objets gérés par des serveurs. L'exploitation orientée-objets permet en outre d'intégrer des réseaux de points de contrôle sur lesquels les communications sont régies par des protocoles différents (le système LONOS se focalise cependant sur la technologie LonWorks et le protocole LonTalk de la société californienne Echelon Corporation).

Le centre du système LONOS est une structure de communication appelée *framework* qui permettra d'une part d'interconnecter les applications (clients) qui doivent exploiter les réseaux et les serveurs qui gèrent les objets et, d'autre part, le *framework* permet un accès distant à ces gestionnaires de réseaux par le biais de réseaux locaux ou d'Internet.

Abstract

The LONOS system is a software entity that allows applications to easily manage control networks in an object-oriented way and that links these control networks with local area networks or Internet.

The object-oriented management allows applications to take account of the very different features of ressources (control equipments) connected to these control networks. The various uses of control equipments (lighting, alarm, and so on) are why these different features are required. In addition to that, control equipments are manufactured by different companies. To take these differences into account, ressources are represented by objects managed by servers. The objet-oriented management allows also different kind of control networks protocol to be used (in this version of LONOS, the only supported protocol is LonTalk from the californian company Echelon Corporation).

The heart of the LONOS system is a communication infrastructure called *framework* that will allow applications (clients) to connect to servers managing objects. On the other side, the *framework* allows remote clients to access objects managed by local servers via local area network or Internet.

Table des matières

Table des figures

Remerciements

La réalisation de ce mémoire et l'implémentation qui en découle ont constitué une activité plaisante car elle touche à deux domaines qui me tiennent à coeur, à savoir l'électronique et l'informatique. Le développement du projet LonOS n'aurait pas été possible sans le concours d'un certain nombre de personnes auxquelles je voudrais, ici, exprimer ma gratitude:

- Monsieur Jean RAMAEKERS, *professeur de systèmes d'exploitation à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix, à Namur.*
- Monsieur Jean-Pierre PETERS, *professeur de théorie du signal aux Facultés Universitaires Notre-Dame de la Paix, à Namur, Project manager de la société DTI*[1] *s.a. à Naninne.*
- Monsieur Frederic PETERS, *ingénieur en électronique, employé par la société DTI s.a. à Naninne.*
- Monsieur Hoan BUI-XUAN, *informaticien, développeur chez Bull, les Clayes-sous-Bois en France.*
- Monsieur Philippe BAUDRY, *informaticien, développeur chez Bull, les Clayes-sous-Bois en France.*
- Monsieur François LEYGUES, *informaticien, responsable du projet* MONITOR *chez Bull, les Clayes-sous-Bois en France.*
- Monsieur Pierre ANDREI, *informaticien, développeur chez Bull, les Clayes-sous-Bois en France.*
- Monsieur Aimé KASSA, *informaticien, assistant à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix, à Namur.*
- Monsieur Thierry KORMAN, *informaticien, un des développeurs du projet Koala Graphics*[2]*, INRIA*[3]*, site de Sophia-Antipolis.*
- Monsieur Philippe LE HÉGARET, *informaticien, un des développeurs du projet KOML*[4]*, INRIA, site de Sophia-Antipolis.*

1. DTI est l'acronyme de *Développement en Traitement de l'Information.*
2. Interface graphique Java destinée à la manipulation d'objets réseau.
3. Institut National de Recherche en Informatique et Automatique, en France.
4. Sérialisation d'objets graphiques Java en XML.

Merci également aux responsables de la société Echelon Corporation pour le CD-ROM de documentation ("LonWorks Networks Reference") qu'ils m'ont gracieusement fait parvenir.

Introduction

L'électronique et les microprocesseurs se retrouvent dans un grand nombre d'équipements aussi bien en industrie qu'à la maison. Ceux-ci remplissent des fonctions diverses telles que le contrôle d'une alaiseuse-fraiseuse, le contrôle du programme d'une machine à laver, de l'alarme d'un immeuble, etc. Lorsque certains de ces équipements sont reliés en réseau et à un ou plusieurs ordinateurs, les possibilités d'applications deviennent quasiment infinies, seule notre imagination est une limite. Ces réseaux, que l'on nommera réseaux industriels ou réseaux domotiques, se distinguent des réseaux locaux reliant nos ordinateurs par certaines caractéristiques:

- Transmission de messages courts;
- Coût d'un noeud très faible. On désignera par le terme noeud une unité de contrôle reliée au réseau industriel ou domotique, c.-à-d. le processeur muni de ses ports de communication et d'entrée/sortie ainsi que les capteurs et actuateurs avec lesquels il s'interface.
- Diversité des supports de communications (réseau propriétaire, ligne RS-485, réseau électrique, etc.);
- Faible bande passante;
- Maintenance faible;
- Diversité de provenance des noeuds.

Parallèlement, l'émergence des nouvelles technologies de l'information et de la communication et de leur plus médiatique matérialisation, Internet, engendre de nouveaux besoins. Les entreprises investissent massivement dans les réseaux locaux, le développement d'applications client-serveur, la création de sites Web, ...

Pourquoi dès lors ne pas connecter ces réseaux industriels et domotiques à Internet ou à un intranet afin de permettre leur exploitation et leur contrôle à distance? Ou encore permettre d'effectuer le lien entre deux réseaux de contrôle distants par le biais d'Internet?

Le système LonOS, le projet personnel dont ce document présente l'analyse, offre une réponse à ces besoins et à d'autres plus spécifiques à la technologie utilisée, la

technologie LonWorks. Dans les chapitres qui suivent, les besoins seront identifiés plus précisément. En outre, le contexte de développement, la société DTI s.a. et le contexte d'inspiration, BULL, seront présentés.

On spécifiera alors les fonctionnalités du système LONOS et le modèle de gestion orienté-objets. Puis les détails d'implémentation seront présentés ainsi que les prérequis nécessaires à la compréhension du tout. Enfin, on utilisera le système ainsi constitué pour le développement d'une application typique à base de réseaux de contrôle.

Pour terminer cette introduction, penchons nous sur le choix du titre de ce mémoire afin d'éviter un éventuel malentendu. Le choix de "système d'exploitation" n'est pas à prendre comme la désignation d'un système d'exploitation au sens système opératoire (un système d'exploitation UNIX, par exemple), mais bien comme une couche logicielle qui permet à des applications d'**exploiter** un réseau ou un ensemble de réseaux de points de contrôle.

Chapitre 1

Le contexte

Le projet LonOS que j'ai développé correspond à une demande de la société DTI s.a. à Naninne, active dans le domaine de l'électronique médicale, du contrôle d'accès et de la domotique. Pour mieux cerner ses besoins, nous allons présenter quelques-unes de ses applications. On pourra également se référer à [1] pour d'autres exemples. Après cette brève présentation, la technologie LonWorks, à laquelle ce système est dédié, sera introduite dans les grandes lignes. Ce chapitre se terminera par la présentation de la suite logicielle OpenMaster (administration de systèmes) de la société française BULL qui fut une source d'inspiration pour le développement de ce projet.

1.1 Des applications

1.1.1 Contrôle de photocopieurs

Une première application basée sur la technologie LonWorks est la réalisation d'un système de contrôle d'accès à des photocopieurs. Chaque photocopieur est muni d'un appareil doté d'un lecteur de carte magnétique. Cet appareil lit le crédit de l'utilisateur sur la carte et autorise l'activation du photocopieur si ce crédit le permet. En outre, à chaque photocopie, le crédit de la carte est diminué. Le responsable de la salle du centre de réprographie a à sa disposition un ordinateur (que l'on appelera par la suite "hôte de gestion") sur lequel il voit une représentation de chacun des photocopieurs avec ses caractéristiques, l'état d'une carte introduite dans l'appareil correspondant, etc. A l'aide de cet ordinateur, le responsable peut définir le coût d'une photocopie pour chaque photocopieur (celui-ci est en effet différent s'il s'agit d'un photocopieur couleur ou noir et blanc). Il a en outre un appareil permettant le rechargement d'une carte magnétique. Cette application est illustrée à la figure 1.1.

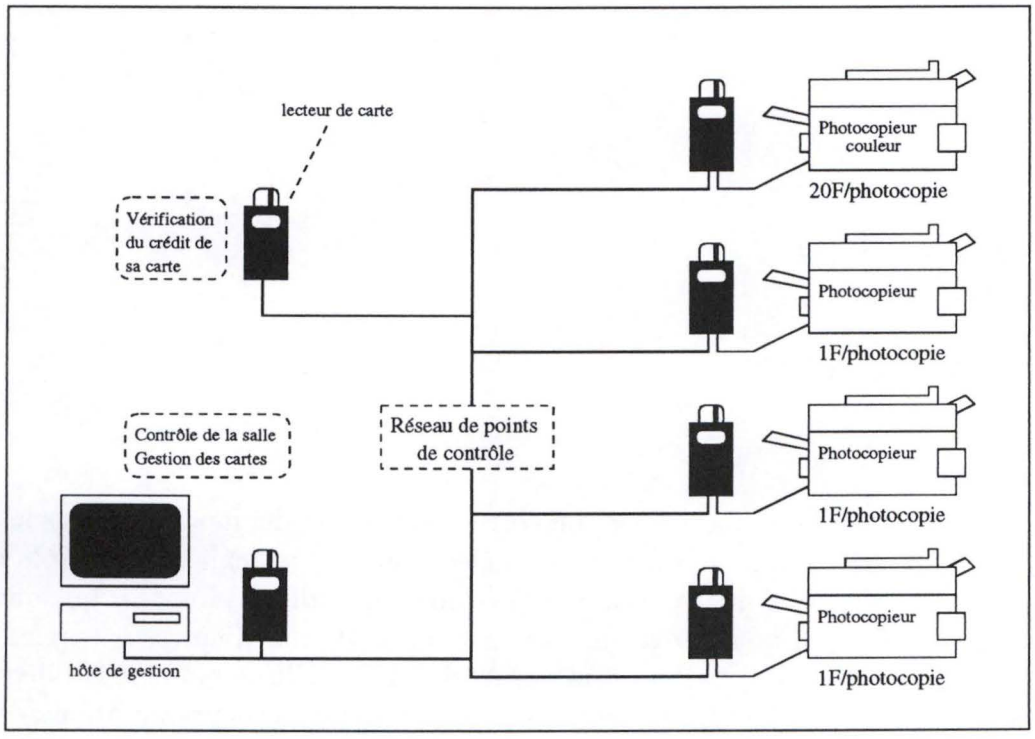

FIG. 1.1 – *Contrôle de photocopieurs*

1.1.2 Contrôle de discothèques

Le contrôle de la distribution des boissons dans les discothèques est également une application de la technologie LonWorks. Ici, chaque bouteille d'alcool est munie d'une électrovanne et d'un débitmètre[1] reliés à un noeud du réseau qui permet la distribution et le contrôle de la dose délivrée. Ces noeuds sont également reliés à une série de caisses enregistreuses et d'ordinateurs qui permettent le contrôle du stock, des factures pour chaque personne responsable de la distribution (cette personne est identifié par une clé électronique qu'il doit présenter à l'appareil avant la distribution d'une boisson). La figure 1.2 illustre cet exemple.

1.1.3 Contrôle d'équipements domotiques

Cette application se veut plus générique que les précédentes (1.1.1 et 1.1.2). On se trouve en face d'une multitude d'équipements aux fonctionnalités diverses et à priori inconnues tels que interrupteurs, lampes, dimmers, détecteurs de fumée, thermostats, volets électriques, alarmes, etc. Dans les applications de gestion de photocopieurs et de

1. Le débitmètre est un appareil destiné à la mesure du débit d'un liquide.

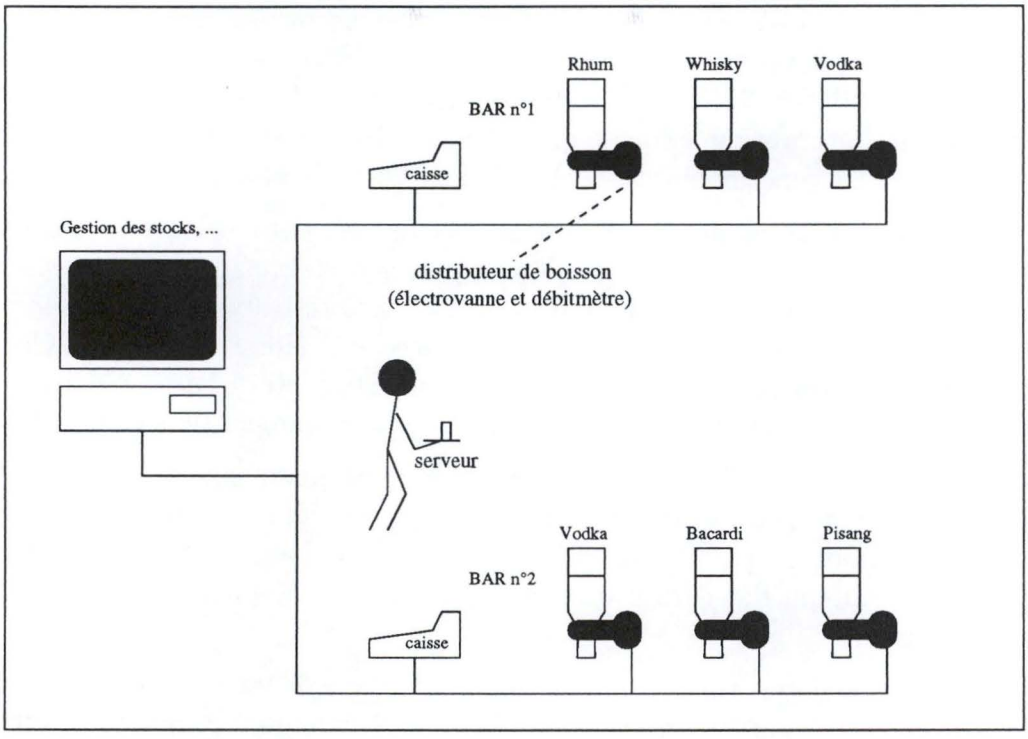

FIG. 1.2 – *Contrôle de discothèques*

discothèques, les noeuds étaient parfaitement connus puisqu'ils étaient conçus spécifiquement pour l'application.

On veut donc disposer dans cette application d'un mécanisme permettant d'intégrer ces équipements divers[2]. On voudrait pouvoir indiquer au système qu'une action sur un noeud entraînera une autre action sur un autre noeud. Par exemple, on voudrait que le basculement en position ON d'un interrupteur déclenche l'allumage d'une lampe. Le basculement en position OFF en déclenche l'extinction.

1.2 La technologie LonWorks

Les applications présentées ci-dessus reposent toutes sur la technologie LonWorks de la société californienne Echelon Corporation. Nous nous baserons sur cette technologie pour développer le système LONOS, mais essayerons de rester le plus général possible afin d'en permettre l'extension future à d'autres standards.

2. On verra plus loin (section 1.2.2) que la technologie LonWorks permet déjà d'apporter une solution partielle à ce problème par l'intermédiaire de variables-réseau. Ce qu'Echelon Corporation dénomme "l'interopérabilité" des points de contrôle.

La technologie LonWorks est le fruit d'une recherche menée conjointement par plusieurs entreprises opérant dans le domaine des semiconducteurs et des réseaux de contrôle dans le but d'améliorer l'interopérabilité de ces derniers. Parmi ces entreprises, on retrouve Echelon Corporation, Motorola, Toshiba ainsi que des collaborateurs tels que Cisco et Philips. Cet ensemble est dénommé "LonWorks Group".

Premièrement, tentons de déterminer quelles ont été les motivations qui ont mené à cette recherche. A l'époque, chacun développait son réseau industriel et ses couches applicatives en fonction de l'usage qu'il en faisait. Les réseaux de contôle de l'époque étaient CAN (Controller Area Network) de Bosch, I^2C (Inter Integrated Circuit) de Philips, ABUS, BitBus, Interbus, le bus de mesure DIN, etc. Mais il n'existait pas de standard en ce qui concernait l'échange de messages, le management et le diagnostic.

La technologie LonWorks est un ensemble de développements matériels et logiciels visant à palier ce manque. Le matériel est constitué d'une série de processeurs spécialisés, de transceivers pour différents supports de communication, ... et le logiciel regroupe le "firmware[3]" contenu dans les processeurs: gestion des communications, gestion des applications, etc.

Les deux sections qui suivent présentent plus en détail les parties matérielle et logicielle. Cependant, pour obtenir les spécifications complètes, il faudra se référer aux documents indiqués pour chacune d'elles.

1.2.1 Architecture matérielle

L'architecture matérielle est basée sur une série de points de contrôle connectés entre eux par un réseau. Chacun des points de contrôle comprend une unité de traitement, des capteurs et des actuateurs ainsi qu'un transceiver permettant la liaison avec le support de communication (voir figure 1.3).

Les informations présentées dans cette section sont extraites du document suivant:

- *LonWorks Technology Devices: MC143150, MC143120, Motorola Inc. 199x. (référence [3])*

Le "LonWorks Group" fournit des composants matériels destinés à faciliter la mise en oeuvre de noeuds LonWorks.

Neuron Chip

Le Neuron Chip est l'unité de traitement d'un point de contrôle. Il s'agit d'un circuit intégré VLSI[4] comprenant trois processeurs(voir figure 1.4), chacun d'eux étant assigné à

3. Firmware: désigne une partie logicielle en général non-modifiable destinée à piloter un équipement matériel.

4. VLSI: Very Large Scale Integration.

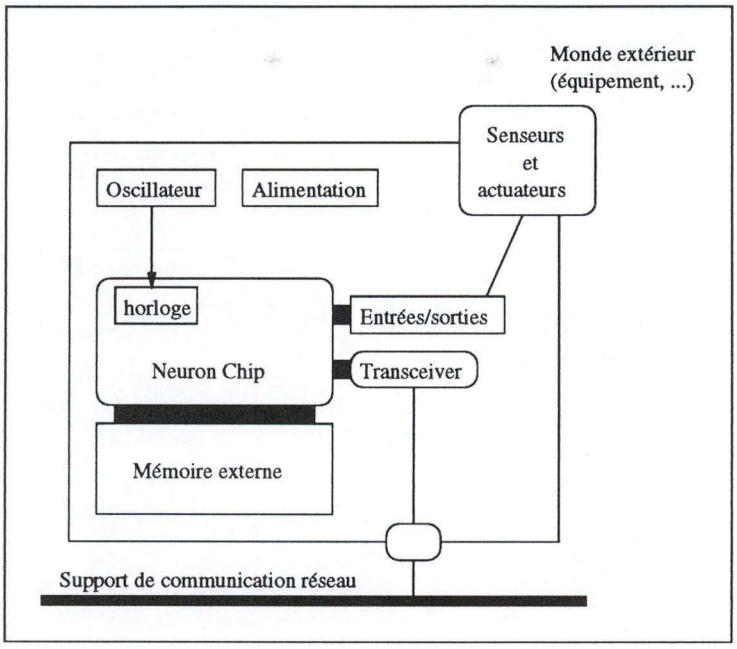

FIG. 1.3 – *Schéma d'un noeud typique.*

une tâche particulière. Ainsi, un Neuron Chip comprend un processeur MAC[5] destiné à la gestion de l'accès au support de communication, un processeur *Réseau* dédié à la gestion des communications suivant le protocole LonTalk (cf. section suivante) et un processeur *Application* qui gère le code applicatif. Bien entendu, un Neuron Chip est également pourvu de mémoire, d'entrées/sorties, d'une horloge et d'un port de communication réseau.

Les trois processeurs du Neuron Chip (*MAC*, *Réseau* et *Application*) communiquent entre eux par l'intermédiaire de blocs de mémoire partagée protégés par des sémaphores matériels (voir figure 1.5). On peut être surpris par le fait que la figure ne mentionne pas de lien entre les processeurs *Application* et *MAC*. Ceci s'explique par le fait que le processeur *Application* s'adresse au processeur Réseau lorsqu'il souhaite émettre une information sur le réseau et c'est le processeur *Réseau* qui va s'adresser au processeur *MAC* pour l'émission de la trame sur le support physique.

Les Neuron Chips contiennent évidemment le logiciel destiné à chacun des processeurs afin qu'ils mènent à bien leurs tâches respectives. On désignera ce logiciel par le terme "firmware" car il n'est pas modifiable par l'utilisateur contrairement à la partie logicielle applicative (software) exécutée par le processeur *Application* et située dans une zone mémoire modifiable par l'utilisateur du noeud (le terme utilisateur désigne en fait

5. MAC: Medium Access Control.

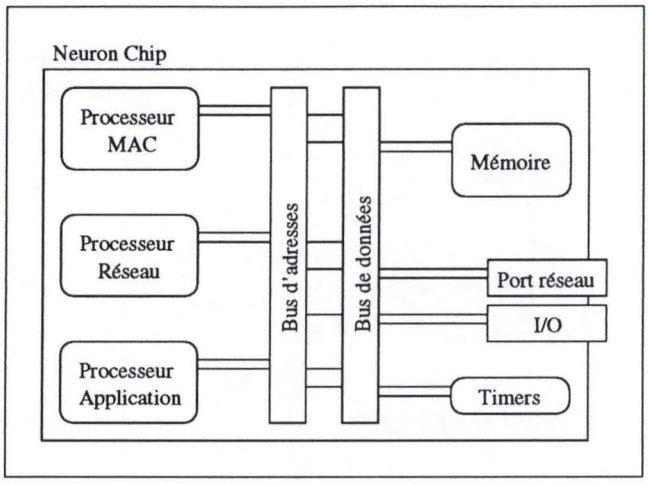

FIG. 1.4 – *Synoptique d'un Neuron Chip.*

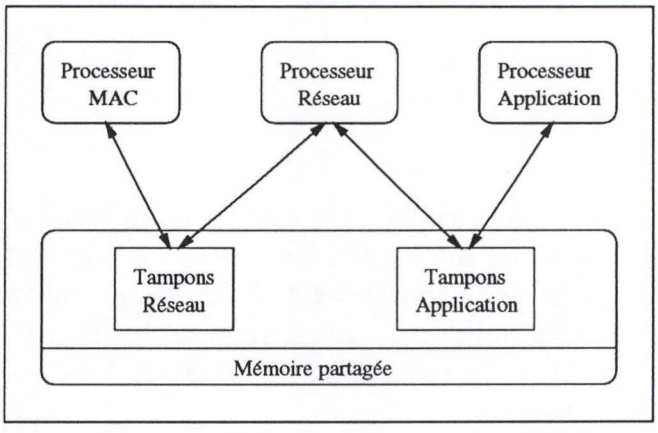

FIG. 1.5 – *Communication entre les processeurs d'un Neuron Chip.*

le développeur de l'équipement reposant sur le Neuron Chip).

Transceiver

Les Neuron Chips sont capables de communiquer entre eux grâce à leur port de communication réseau. Cependant, il est nécessaire de disposer de transceivers pour supporter l'un ou l'autre encodage, support de communication, etc.

1.2.2 Le protocole LonTalk

Objectifs

Le protocole LonTalk régit l'échange de messages entre les applications exécutées par les processeurs *Application* des Neuron Chips. Le protocole LonTalk est destiné aux réseaux de contrôle caractérisés par la transmission de messages courts (quelques octets), le coût relativement faible des noeuds, la diversité des supports de communication, la faible bande-passante, la diversité de provenance des équipements.

Les informations présentées ci-dessous sont extraites des documents suivants et complèteront donc la brève présentation du protocole (on y trouve notamment des schémas explicitant les relations entre les couches du protocole ainsi que les services fournis):

- *LonTalk Protocol Specification, version 3.0, Echelon Corporation, 1994. (référence [4])*
- *LonWorks Engineering Bulletin: LonTalk Protocol, Echelon Corporation, 1993. (référence [5])*

Terminologie

Dans les documents d'Echelon Corporation, le protocole est présenté suivant la terminologie OSI (Open Systems Interconnection). Le protocole est organisé en une pile de couches. Chaque couche fournit un service bien précis et est décrite par ses primitives et les données qui y sont échangées. On désignera par N-PDU une entité d'information échangée entre deux entités protocolaires de la couche N.

Le découpage standard OSI comprend les couches suivantes (voir figure 1.6): Physique, Liaison, Réseau, Transport, Session, Présentation et Application. Le protocole LonTalk suit cette découpe.

Couche Physique

La couche physique gère la transmission et la réception de données binaires. Ces communications peuvent se faire selon deux modes: le mode direct et le mode spécial. Le mode direct utilise l'encodage différentiel de Manchester. Dans le mode spécial, les données sont transmises et reçues sériellement, sans encodage (ce mode est prévu pour l'utilisation d'un transceiver externe au Neuron Chip).

Sous-couche MAC

Cette sous-couche fait partie de la couche Liaison.

La sous-couche MAC fournit l'accès à la couche physique avec des options telles que la priorité d'une trame et/ou la détection/résolution de collision. Le protocole d'accès au

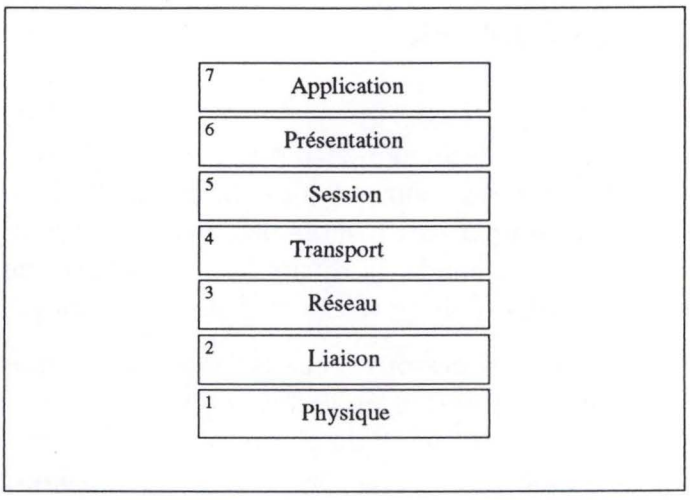

FIG. 1.6 – *Découpage OSI en couches.*

support est appelé "Predictive p-persistent CSMA [6]". Il s'agit d'une technique permettant d'éviter les collisions en accédant au support séquentiellement et en tenant compte de la charge espérée du support. Un CRC[7] de 16 bits est généré à la transmission et vérifié à la réception.

Couche Liaison

La couche Liaison permet la transmission de trames (NPDU) avec les garanties suivantes: ordre, pas d'accusé de réception, détection d'erreur (basé sur CRC) sans récupération.

Couche Réseau

Avant de présenter les services fournis par cette couche, il est impératif de présenter les types d'adressages utilisables par le protocole.

Une adresse, suivant le protocole LonTalk, peut être décrite comme suit:

```
Address ::= (Domain, SubNet, Node)
          | (Domain, SubNet, NeuronID)
          | (Domain, Group, Member)
```

- Le champ *NeuronID* est un identifiant, unique au monde, d'un Neuron Chip. Cet identifiant est inscrit dans le Neuron Chip lors de sa fabrication.

6. CSMA: Carrier Sense Multiple Access.
7. CRC: Cyclic Redundancy Code.

- Le champ *Domain* (domaine) est un groupe de noeuds au sein duquel peuvent avoir lieu les communications suivant le protocole LonTalk. Il n'est pas possible de communiquer d'un domaine à l'autre sans "routage applicatif" (celui-ci peut être effectué par l'application exécutée par le noeud). Un noeud peut appartenir à au plus deux domaines.
- Le champ *SubNet* (sous-réseau) est un groupe de noeuds au sein d'un domaine. Un domaine peut contenir jusqu'à 255 sous-réseaux. Un noeud peut appartenir à deux sous-réseaux au plus et ces deux-ci doivent obligatoirement appartenir à des domaines différents.
- Le champ *Group* est un groupe de noeuds utilisé pour effectuer du multicasting. Un domaine peut contenir jusqu'à 256 groupes.
- Le champ *Node* identifie un noeud au sein d'un sous-réseau.
- Le champ *Member* identifie un noeud au sein d'un groupe.

La couche Réseau fournit un service de transmission et de réception. La transmission s'effectue vers 1 noeud (unicasting), N noeuds (multicasting) ou tous les noeuds d'un domaine (broadcasting), sans connexion et avec les garanties suivantes:

- pas d'accusé de réception;
- pas de re-transmission;
- préservation de l'ordre (hypothèse: topologie sans boucle);
- pas de segmentation.

Couche Transport

La couche transport fournit un service de transmission de bout en bout ("end-to-end") fiable. Cette transmission peut être de type unicast ou multicast. Deux qualités de service sont disponibles:

- Unicast et multicast fiables: Les services fiables offrent les garanties suivantes: (i) transmission fiable avec "best-effort" déterminé par le nombre de tentatives; (ii) détection de paquets dupliqués et (iii) préservation partielle de l'ordre (c.-à-d. que l'ordre est toujours préservé, mais si un paquet n'est pas transmis après un certain nombre de tentatives, le message sera "oublié").
- Unicast et multicast répétés, sans acquittement: Ici, aucun accusé de réception n'est attendu. La transmission du message est répétée un certain nombre de fois.

Couche Session

La couche session fournit un service de requête/réponse destiné à faciliter les communications entre applications. Le service peut s'apparenter à un service d'appel de procédure à distance (RPC — Remote Procedure Call, voir [19]).

La couche session fournit également un service d'authentification. Ce service est basé sur des clés d'authentification partagées par deux noeuds qui doivent s'échanger des messages authentifiés. Ceux-ci sont utilisés pour éviter un accès non autorisés aux noeuds et à leurs applications. Comment cela fonctionne-t-il?

La technique, similaire au protocole d'authentification avec fonction cryptographique à sens unique dans lequel l'authentification a lieu après réception du message (cf. [27]), est la suivante: Quand un noeud R reçoit un message M, il demande au noeud S, l'émetteur de celui-ci, de s'authentifier. Il lui fournit à cet effet un nombre aléatoire C (pour "*challenge*"). S effectue donc une transformation de C avec sa clé K. Appelons le résultat de cette transformation $C' = MAC_k(C)$[8]. C' est alors transmis à R qui a également effectué cette transformation de son côté. Si C' et le résultat de la transformation effectuée par R sont identiques, alors R considère que le message M est acceptable.

Remarquons que la transformation effectuée par K sur C, MAC_k, est telle que même en connaissant C et K', il est extrêmement difficile de déduire K. Le choix d'un nombre aléatoire pour C permet d'empécher le "rejeu".

Couches Présentation et Application

Ces couches fournissent un service d'échange de messages qui se décline en services spécialisés énumérés ci-dessous::

- Propagation de variables-réseau,
- Passage de messages "génériques" (c.-à-d. que leur signification est connue de l'application du noeud),
- Gestion du réseau (avec des messages spécifiques),
- Diagnostic du réseau (avec des messages spécifiques),
- Transmission de trames étrangères.

Les variables-réseau sont des variables gérées par le code applicatif contenu dans les Neuron Chips. Ces variables peuvent être connectées entre elles grâce au mécanisme de propagation de variables-réseau (une illustration en est donnée à la figure 1.7). Ce mécanisme consiste à déterminer un ensemble de variables (situées sur des noeuds différents en général) et à en faire un groupe. Dans le groupe, il y a des variables d'entrée et des variables de sortie. Si une variable d'entrée du groupe est modifiée, les variables de sortie de cette connexion sont également modifiées.

Cette modification des variables de sortie par une variable d'entrée est réalisée par le mécanisme de propagation de variables-réseau qui consiste, pour le noeud qui détient la variable d'entrée concernée, à émettre un message de propagation de variable vers les noeuds qui détiennent les variables de sortie de sorte qu'ils puissent effectuer la mise à jour de celles-ci.

8. MAC désigne ici Message Authentication Code, également appelé certificat.

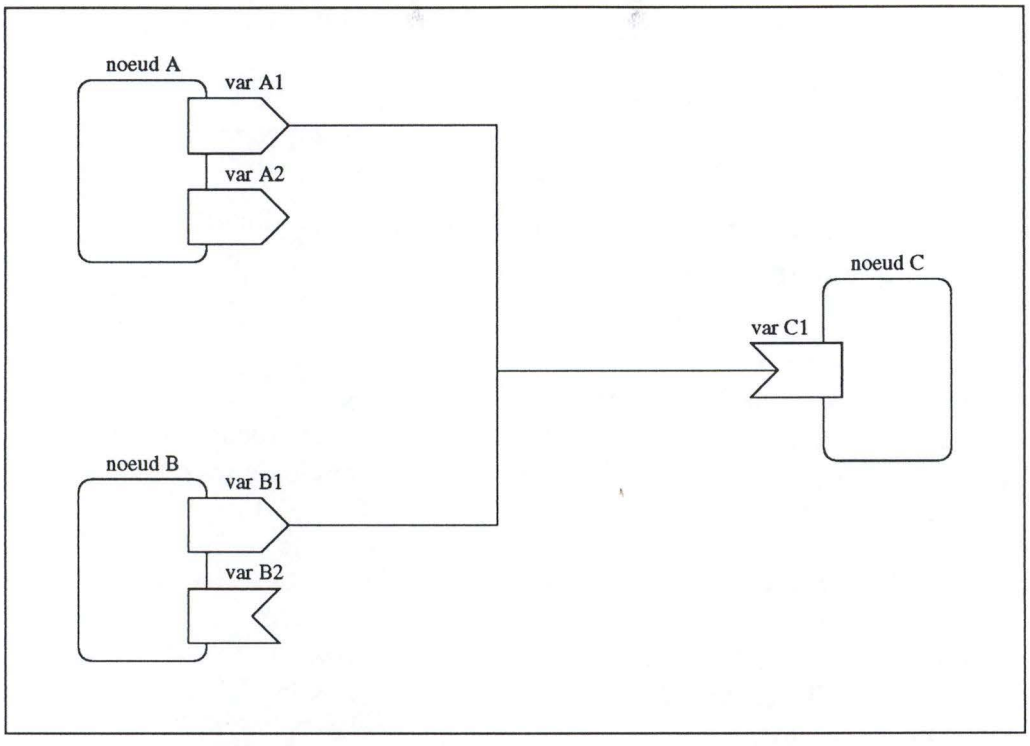

FIG. 1.7 – *Exemple de connexion de variables.*

Remarquons que le rôle exact de la couche Application dans le mécanisme des variables-réseau est de garantir la compatibilité entre les applications exécutées par les différents noeuds. Cette compatibilité est rendue possible par des types de variables-réseau standard (ceux-ci sont appelés SNVT — Standard Network Variable Types — par Echelon Corporation). Mentionnons encore que les variables-réseau sont définies par le code applicatif (créé par le développeur du noeud). Lorsque ce code (écrit en *Neuron C*) est compilé, un fichier d'information décrivant les variables-réseau (notamment le lien entre le nom de la variable dans le code et l'index l'identifiant au sein du *Neuron Chip* ainsi que le type de la variable) est produit. Celui-ci, de type XIF — eXternal Information File —, est nécessaire à la configuration du noeud, à la création de connexions de variables, etc. Nous y ferons plusieurs fois référence par la suite.

Le service de transmission de messages génériques consiste à transmettre des messages dont le sens est défini par le développeur. La signification de ces messages sera connue des seules applications prévues pour la maniuplation de ceux-ci.

Les couches Présentation et Application offrent égalemment des services de gestion et de diagnostic du réseau. Ces services sont basés sur la transmission de messages spécifiques. Les services de gestion et de diagnostic du réseau offrent, entre autres, les

possibilités suivantes:

- Configuration des adresses, domaines;
- Configuration des clés d'authentification (voir couche session);
- Installation d'un programme applicatif;
- Configuration des variables-réseau (création de connexions, etc.);
- Interrogation du noeud (état et statistiques);
- Remise à zéro.

Le service de transmission de trames étrangères permet l'encapsulation de trames étrangères dans un message LonTalk et sa transmission d'un noeud à l'autre. On pourrait également appeler ce service "LonTalk tunneling" par analogie avec "l'IP tunneling".

1.3 La gestion de réseau chez BULL

Dans le cadre de cette année académique, j'ai eu l'occasion d'effectuer un stage de six mois sur le site des Clayes-sous-Bois de la société française BULL s.a. dont les domaines d'activité sont entre autres

- les cartes à puces (CP8),
- les portables,
- la sécurité des systèmes,
- télécommunications,
- les distributeurs de billets (DAB[9]/GAB[10]).

La tâche qui m'incombait s'inscrivait dans le développement de la suite logicielle OPENMASTER (dont on peut trouver le schéma général à la figure 1.9). Celle-ci est dédiée à l'administration des systèmes et réseaux d'un système d'information (cf. [6]).

Mon rôle consistait à développer en Java une interface graphique basée sur des composants, destinée à contrôler des objets administrables et à mener un certain nombre d'actions sur ceux-ci. J'ai donc pu me pencher sur la façon dont OPENMASTER concevait l'administration de systèmes. Les principes mis en oeuvre sont ceux décrits par OSI. Ces principes sont brièvement présentés ci-dessous, ce qui me permettra par la suite d'effectuer certaines analogies avec la gestion de réseaux de contrôle.

9. Distributeur Automatique de Billets
10. Guichet Automatique de Banque

1.3.1 Principes de l'administration de systèmes

L'administration de systèmes selon OSI (tiré de [6]) comprend les tâches qui suivent. OSI prévoit des procédures standardisées pour la plupart d'entre elles.

- Gestion des fautes:
  - Détecter les fautes;
  - Enregistrer les événements (log);
  - Prévoir et exécuter des tests, trouver l'origine des fautes et corriger.
- Gestion des comptes:
  - Gérer les limites par utilisateur;
  - Comptabiliser les consommations de ressources d'un utilisateur;
- Configuration et gestion des noms
  - Collecter les informations concernant l'état actuel des ressources;
  - Définir et modifier les paramètres des composants du réseau;
  - Initialiser et arrêter les objets administrés;
  - Changer la configuration;
  - Associer des noms à des objets et ensembles d'objets.
- Gestion des performances
  - Collecter les informations concernant le niveau de performance actuel des ressources;
  - Maintenir et examiner les enregistrements de performances à des fins de planification et de statistiques.
- Gestion de la sécurité
  - Déterminer les permissions;
  - Contrôler les accès;
  - Gérer le chiffrement et les clés;
  - Authentifier;
  - Enregistrer les événements.

Le modèle d'administration de systèmes selon OSI est représenté à la figure 1.8 (voir [6] pour une description détaillée).

L'administration repose sur l'utilisation d'une série d'entités logicielles (SMAP — Systems Management Application Process) de deux types. Premièrement les entités logicielles desquelles sont issues les commandes d'administration. Elles sont dénommées gestionnaires (SMAP-managers). Et deuxièmement, les entités logicielles attachées à une ou plusieurs ressources administrées. Celles-ci ont le rôle d'agent (SMAP-agents) et sont

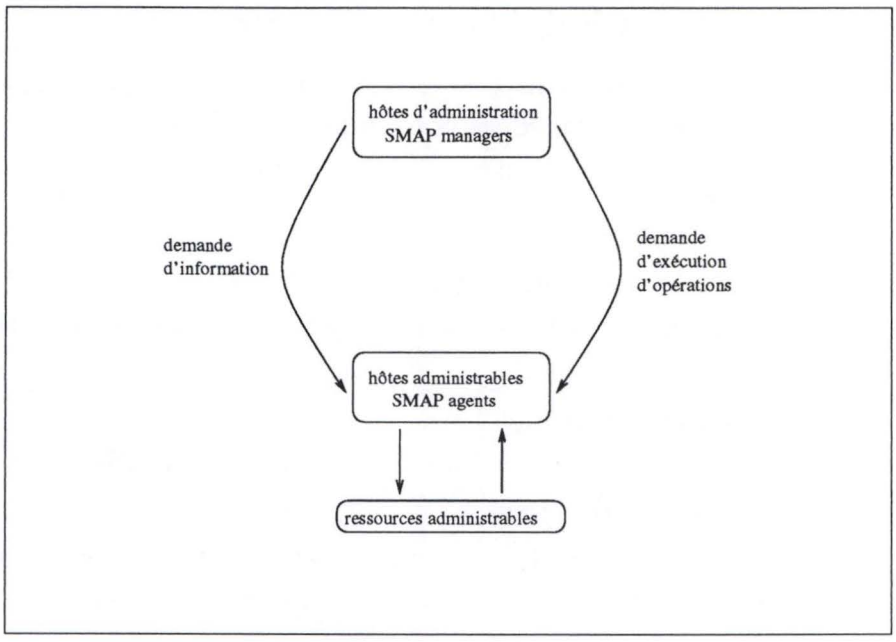

FIG. 1.8 – *Modèle général de l'administration de systèmes.*

chargées d'accéder aux paramètres des ressources et d'exécuter certaines opérations sur celles-ci.

Les communications entre les SMAP-managers et les SMAP-agents sont soumises à la standardisation et donc régies par un protocole bien défini (il existe une série de protocoles d'administration tels que SNMP, CMIP). Ces communications consistent en l'envoi de commandes (par exemple, interrogation d'un paramètre d'une ressource) des SMAP-managers vers les SMAP-agents et de réponses (par exemple, renvoyer la valeur d'un paramètre) ou de notifications (par exemple, indiquer un malfonctionnement, ou un événement particulier) de la part de ces derniers.

La façon dont un agent accède aux paramètres des resources dont il est "responsable" n'est pas soumise à la standardisation. Cependant, cet agent doit publier une série d'informations qui décrivent précisément les ressources qui sont administrables par son biais. Cette série d'information est appelée base d'information de gestion ou MIB, en anglais (Management Information Base). Ainsi, un SMAP-manager peut déterminer quelles sont les ressources que gère un SMAP-agent, en s'adressant à celui-ci.

1.3.2 La suite logicielle OPENMASTER

OPENMASTER est basé sur une architecture ouverte et orientée-objets conçue pour des environnements hétérogènes. Ceci permet d'administrer tout matériel supportant un

protocole d'administration tel que SNMP[11] (IETF[12]), CMIS/CMIP[13] (OSI), DSAC[14] (BULL), SNA[15] (IBM), etc.

OPENMASTER supporte, en outre, de multiples plate-formes et est disponible sur les plate-formes suivantes: Windows NT (Microsoft), AIX (IBM), Solaris (Sun), SGI, HP. Bull annonce par ailleurs sa disponibilité prochaine pour le système d'exploitation Linux.

Du fait que OPENMASTER est un produit ouvert, son adaptation à de nouveaux types de matériel à administrer et le développement spécifique de serveurs métiers et d'applications d'administration sont aisés.

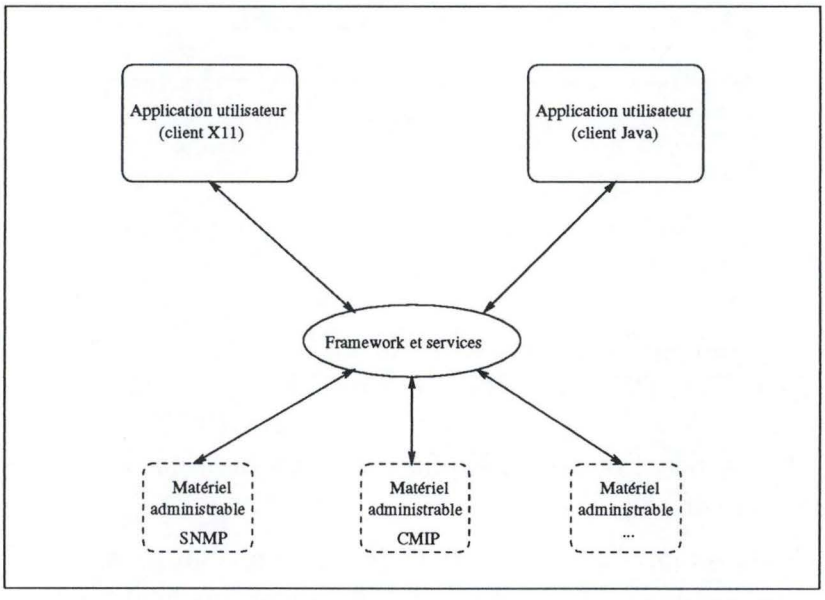

FIG. 1.9 – *Schéma général de la suite logicielle* OPENMASTER.

L'architecture d'OPENMASTER est représentée à la figure 1.10. Les différents éléments composant le système sont décrits ci-dessous:

Les agents sont des unités logicielles qui fournissent un certain nombre d'informations d'administration. Ils sont en général fournis par le constructeur du système ou du matériel et peuvent être partie intégrante du matériel (par exemple d'un routeur).

11. SNMP: Simple Network Management Protocol
12. IETF: Internet Engineering Task Force
13. CMIS/CMIP: Common Management Information Service/Protocol
14. DSAC: Distributed Systems Administration and Control
15. SNA: Systems Network Architecture

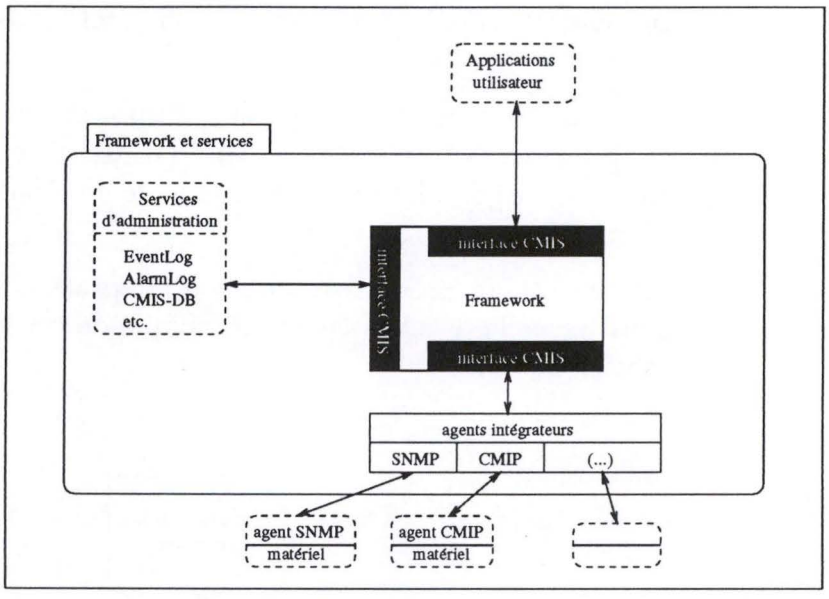

FIG. 1.10 – *Architecture d'*OPENMASTER.

Le framework (infrastructure de communication) permet la gestion de tous les messages CMIS (CREATE, DELETE, RESPONSE, SET, GET, ACTION et NOTIFICATION) et leur transmission vers les services ou applications abonnés. On le nomme également CMIS dispatcher. Pour obtenir plus d'informations sur le framework, on pourra se référer au document [20].

Les agents intégrateurs ont pour rôle la traduction d'un protocole d'administration particulier vers le protocole CMIP. Il s'agit donc d'une couche d'abstraction très importante.

Les services d'administration sont des applications qui permettent le stockage d'alarmes ou d'événements (AlarmLog ou EventLog), la gestion de bases d'inventaires (CMIS-DB), la gestion des performances, la découverte du réseau, la corrélation d'alarmes, etc.

1.4 Conclusion

Le contexte de développement du projet LONOS est donc la société DTI et ses applications reposant sur la technologie LonWorks de la société californienne Echelon Corporation. Cette technologie est développée dans le but de faciliter l'interopérabilité des réseaux de contrôle. De ce contexte, on va tirer les éléments nécessaires à l'exploitation d'un réseau de points de contrôle (au chapitre 2).

L'application OPENMASTER de la société française BULL constitue un exemple de

système d'administration reposant sur les principes d'OSI. Ces principes sont cependant relatifs aux systèmes d'information à base de réseaux d'ordinateurs. Néanmoins, le système LONOS s'inspire de l'architecture distribuée d'OPENMASTER. On verra plus loin les modifications qui ont été nécessaires pour répondre aux besoins particuliers des réseaux de contrôle.

Chapitre 2

Les besoins

2.1 Introduction

Dans ce chapitre, nous allons essayer de répertorier les besoins associés à l'exploitation d'un ou plusieurs réseaux de points de contrôle, au vu de la connaissance du contexte que nous avons acquise au chapitre précédent.

Les besoins repris ci-dessous sont groupés en trois catégories. La première, *Les besoins minimums*, décrit ce qui est absolument nécessaire pour accéder au réseau. La seconde partie, intitulée *Les besoins d'exploitation*, présente les fonctionnalités nécessaires à une exploitation d'une part plus générique et d'autre part plus extensible (ce qui fait l'intérêt du système LONOS). Enfin, des besoins auxquels une réponse n'a pas été apportée à l'heure actuelle seront mentionnés sous la section *Autres besoins*.

2.2 Besoins minimums

2.2.1 Accès au réseau

La fonctionnalité minimale que l'on attend d'un système d'exploitation est celle qui fournit un accès aux ressources gérées. En l'occurence, il s'agit du ou des réseaux LonTalk ainsi que les noeuds qui y sont connectés.

L'accès au réseau est fourni par une interface matérielle connectée d'une part à l'hôte de gestion et d'autre part au réseau LonTalk. Cette interface peut prendre plusieurs formes. Par exemple, la société DTI propose une carte ISA 16 bits à insérer dans un PC ou un adaptateur RS-232 vers LonTalk.

Quelle que soit l'interface matérielle dont on dispose, celle-ci nécessite un gestionnaire ou pilote de périphérique (qui sera manipulé par le système d'exploitation — operating system — de la machine hôte). Il s'agit donc du besoin minimum nécessaire

à l'exploitation du ou des réseaux LonTalk. Dans le cas particulier de la société DTI, ce pilote de périphérique existe actuellement pour le système d'exploitation Windows 95 de Microsoft.

2.2.2 Support du protocole

Une fois que l'on a accès aux réseaux LonTalk (par le biais des interfaces matérielles et de leurs pilotes), il faut être capable de "parler une langue compréhensible par les noeuds connectés à ces réseaux LonTalk". Ceci signifie que le système d'exploitation (LONOS) doit supporter le protocole utilisé sur le réseau. Dans le cas de la technologie LonWorks, il s'agit du protocole LonTalk. Cette partie du LONOS chargée de supporter le protocole LonTalk sera appelée *gestionnaire de réseau* dans la suite du document.

Ce gestionnaire de réseau devra, en particulier, tirer parti des services offerts par les couches 6 et 7 (présentation et application) du protocole LonTalk. Ces couches permettent notamment la configuration des noeuds, l'interrogation des noeuds sur leur état, la propagation de variables-réseau, etc. (voir la figure 1.2).

A ce niveau, c.-à-d. une fois que l'on a accès au réseau et que l'on peut "parler LonTalk", on doit disposer d'une plate-forme dont la structure est illustrée à la figure 2.1.

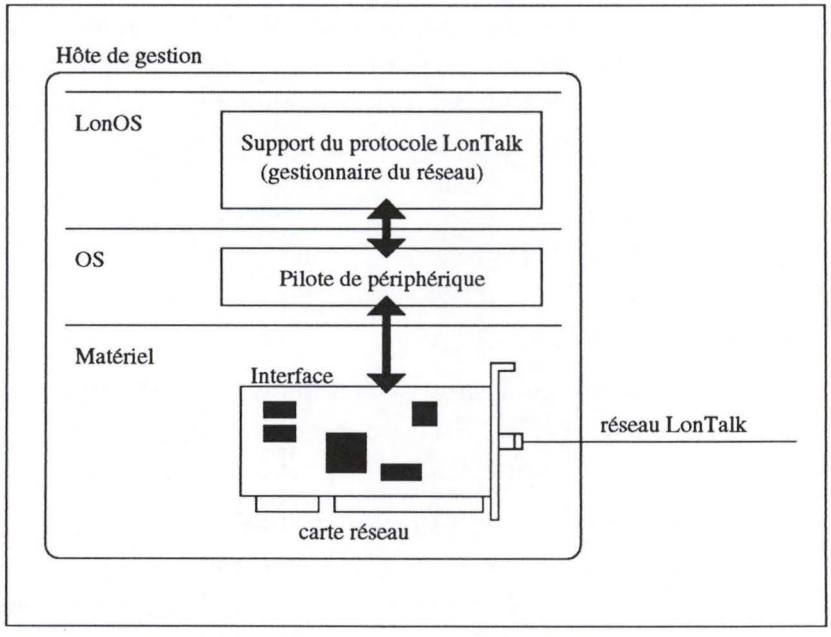

FIG. 2.1 – *Hôte de gestion répondant aux besoins minimums.*

Cette couche logicielle (support du protocole LonTalk) existe également chez DTI, mais souffre d'un certain nombre de restrictions. Elle se présente sous la forme d'une librairie partagée Windows (DLL) chargée par l'application qui la nécessite (c.-à-d. l'application qui doit accéder au réseau). A l'aube du développement du LONOS, les limitations de cette librairie étaient les suivantes:

- Elle ne peut être utilisée par plusieurs applications simultanément;
- Elle ne gère qu'une seule interface (et donc un seul réseau);
- Elle a été développée pour des applications spécifiques telles que celles qui furent présentées dans le chapitre 1. Elle ne permet donc pas l'exploitation de toutes les fonctionnalités de la technologie LonWorks et du protocole de communication LonTalk.

Des efforts ont récemment été entrepris par DTI afin de permettre à plusieurs applications d'utiliser la librairie simultanément.

2.3 Besoins d'exploitation

2.3.1 Gestion orientée-objets

Dans un souci d'extensibilité et de généricité, on souhaite que les gestionnaires de réseau LonTalk dont on a parlé précédemment puissent présenter les éléments du réseau qu'ils gèrent sous la forme d'entités abstraites, c.-à-d. des objets. Typiquement, les objets gérés sont les noeuds du réseau. Dans le cas de la technologie LonWorks, on pourra également représenter des éléments qui n'ont pas une existence physique mais logique comme les connexions de variables. Enfin, on pense à la création d'objets virtuels à des fins de simulation ou de monitoring.

2.3.2 Multi-clients, multi-serveurs

On souhaite que le système dont on établit les besoins soit organisé selon le modèle client-serveur. Ceci signifie que le gestionnaire de réseau dont on a parlé ci-dessus et qui gère des objets représentant les noeuds et connexions du réseau soit un serveur auquel des clients vont pouvoir adresser des requêtes.

En outre, on peut avoir de multiples serveurs et de multiples clients (voir figure 2.2). Ceci est la conséquence des constatations qui suivent: (i) le système LONOS doit pouvoir gérer une multitude de réseaux; (ii) le système LONOS doit pouvoir abriter d'autres services que la gestion de réseaux LonTalk et (iii) le système LONOS peut être partagé par plusieurs clients (en général, les applications-utilisateur).

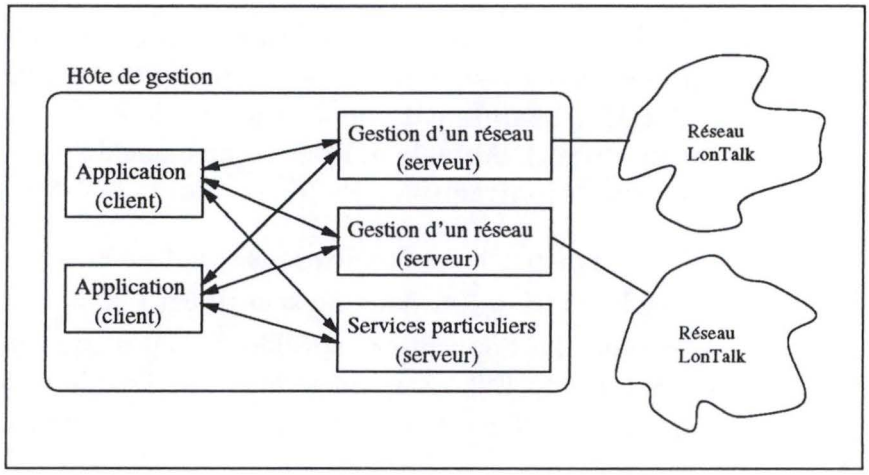

FIG. 2.2 – *Multiples clients et multiples serveurs.*

Multi-réseaux: Dans la plupart des cas, on ne gèrera pas un unique réseau LonTalk, mais un ensemble de réseaux avec éventuellement des interfaces (matérielles) et des protocoles (si on ne se limite pas à LonTalk) différents. Il faut donc avoir la possibilité d'intégrer, au sein du système LONOS, plusieurs gestionnaires de réseau.

Autres serveurs: Dans le cadre de l'ouverture et de l'extensibilité du système LONOS, on souhaite y intégrer d'autres services (non prévus au départ, par exemple). Ces services sont rendus par d'autres serveurs que les gestionnaires de réseau. On pense à un serveur qui constitue une interface avec une base de données, ou qui fournit des services de plus haut niveau que la gestion de réseaux. Par exemple, dans le cas de la gestion d'un réseau domotique, des noeuds sont dédiés à la mesure de la température de plusieurs pièces d'un immeuble et d'autres noeuds actionnent les vannes du chauffage de chaque pièce. On peut envisager un serveur auquel on soumet la température demandée pour chaque pièce et qui par un système d'optimisation et d'apprentissage (basé, pourquoi pas, sur un réseau de neurones) détermine dynamiquement la position des vannes du chauffage pour chaque pièce. Il s'agit bien d'un service de plus haut niveau que la gestion du réseau. En outre, le serveur que l'on vient de décrire est lui-même client des gestionnaires de réseaux car il doit s"adresser à ceux-ci pour accéder aux noeuds.

Multi-clients: On peut retrouver sur un même réseau LonTalk ou sur un ensemble de réseaux gérés par un même système LONOS des noeuds ayant des fonctionnalités diverses. Par exemple, si on reste dans le cadre de l'application de domotique, on pourra retrouver des noeuds relatifs à l'éclairage, d'autres destinés au contrôle du chauffage et enfin des noeuds qui font partie du système d'alarme. Ces trois catégories de noeuds

peuvent très bien être manipulées par des applications-utilisateur différentes. Ce qui implique qu'il nous faille la possibilité de faire cohabiter plusieurs clients.

2.3.3 Accès distant

Il existe une multitude de cas où les applications sont exécutées sur une machine éloignée de celle qui gère le réseau LonTalk. Les deux machines (l'hôte de gestion et celle qui exécute l'application-client) peuvent se trouver dans des pièces différentes voire sur des sites différents. Il faut bien entendu que celles-ci soient reliées par un moyen de communication (on pense à un réseau local ou une ligne téléphonique).

Lorsque l'on parle de machines interconnectées, on pense directement à Internet et au protocole TCP/IP (figure 2.3). On souhaite que le système LonOS permette à des applications et des serveurs distants de communiquer entre eux pourvu que les machines sur lesquelles ils sont exécutés soient reliées par un moyen de communication et que le protocole TCP/IP puisse être utilisé (ceci relève généralement du système d'exploitation des machines).

2.3.4 Anonymat

La question qui se pose maintenant est la suivante: comment un client sait-il sur quel serveur se trouve l'objet auquel il veut accéder? La première solution consiste à indiquer au client à quel serveur il doit s'adresser. Mais, si, pour une raison ou une autre, cet objet est déplacé, il faudra l'indiquer au client. On souhaite que le client n'ait pas à se préoccuper de savoir où se trouve l'objet auquel il veut accéder. C'est le concept d'anonymat (cf. [9]) ou de transparence.

2.4 Autres besoins

Les besoins repris dans cette section sont également importants, mais il ne seront pas traité dans le reste de ce document et ils ne sont pas encore implémentés dans la version actuelle du système LonOS. Cependant, il est bon de les présenter:

2.4.1 Sécurité

On souhaite pouvoir contrôler l'accès au système ainsi que restreindre les opérations possibles sur certaines ressources et ceci sur base d'une identification de l'utilisateur. Les besoins en matière de sécurité se développent donc en

- Acceptation de connexion si l'adresse IP est validée (vérifier l'adresse IP d'un client ou d'un serveur qui se connecte au système);

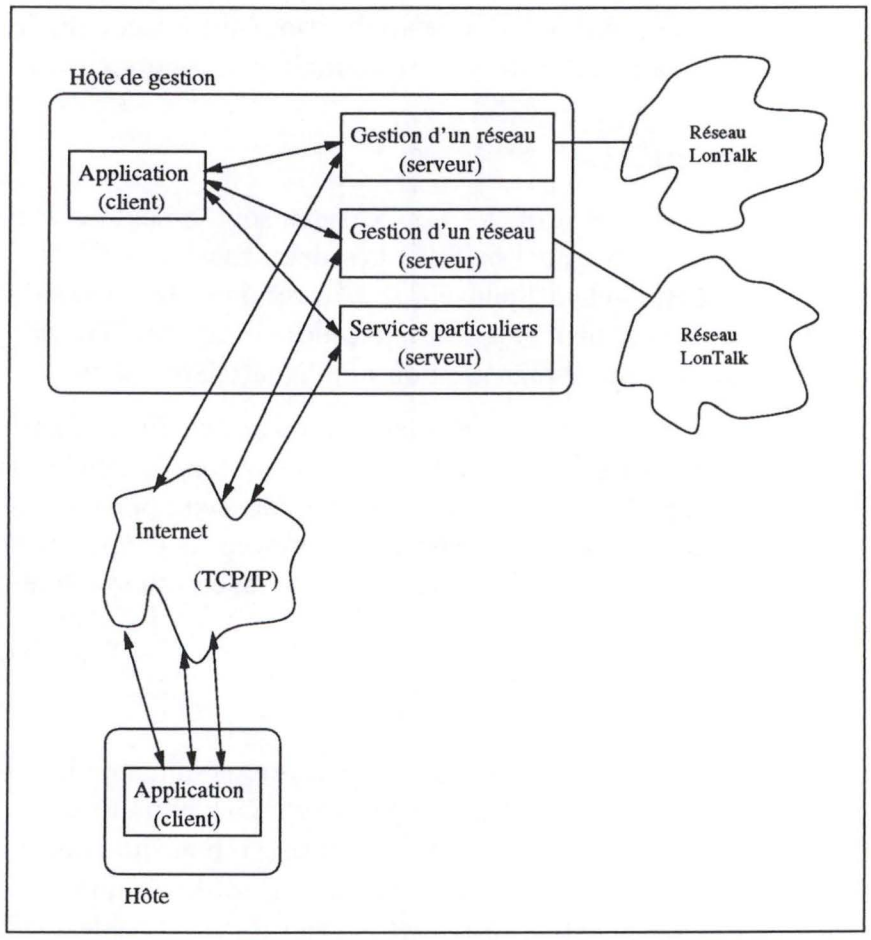

FIG. 2.3 – *Accès distant.*

- Chiffrement des messages échangés (il faut aussi pouvoir empêcher le problème du "rejeu");
- Identification des utilisateurs et notion de rôle: quelles sont les opérations qu'un utilisateur qui s'est identifié (authentifié) peut effectuer sur un objet administrable?

2.4.2 Interface ergonomique

On vient d'indiquer que l'on souhaite représenter des entités physiques ou logicielles par des objets. Il est intéressant, pour pouvoir administrer ces objets, d'en obtenir une représentation graphique, manipulable à la souris. On pourrait par exemple accéder directement aux attributs d'un objet et les éditer, etc.

Par exemple, la suite logicielle OpenMaster de Bull fournit une interface de ce type, destinée au monitoring des objets administrables. Ceux-ci sont représentés sous forme d'icônes manipulables à la souris. A chacun de ces objets graphiques peuvent être associés des composants utilisés pour afficher l'état d'un attribut (par exemple, le nombre de paquets IP reçus ou émis). En outre un menu contextuel permet la configuration aisée de tous ces objets. Il existe deux versions de cette interface destinées à la plate-forme X11 Unix et au Web (sous la forme d'une applet Java dont la partie graphique — Koala — est développée par l'INRIA).

2.4.3 Gestion de transactions

Un autre point important qui ne sera pas développé dans ce document concerne la possibilité de grouper une série de requêtes au sein d'une transaction et de garantir l'atomicité de celle-ci (cf. [19]). Un mémoire complet pourrait être consacré à l'analyse et le développement de cette unique partie. Il s'agit bien d'une gestion de transactions dans un système distribué. On pourra utiliser le protocole "2 phases commit protocol" distribué (voir [19]).

2.5 Conclusion

Ce chapitre présentait les besoins relatifs à l'exploitation de réseaux de points de contrôle. Les besoins suivants seront étudiés dans les chapitres qui suivent:

1. Gestion orientée-objets (traité au chapitre 3),
2. Multi-clients et multi-serveurs (traité au chapitre 4),
3. Accès distant (traité au chapitre 4),
4. Anonymat (traité au chapitre 4).

En marge des grandes nécessités que l'on vient de décrire, se trouvent des besoins annexes tel que l'asynchronisme et la portabilité du système:

Asynchronisme: Lorsqu'un client émet une requête vers le framework, il doit attendre que le message soit transmis (ceci peut prendre un temps non négligeable sur une ligne téléphonique), que le message soit routé, que le serveur concerné par le message réponde à la requête puis que le framework route la réponse vers le client.

On souhaite que l'émission d'une requête par un client ne soit pas bloquante. Dès qu'il a effectué l'appel d'émission, le processus reprend la main. Il devra ensuite s'inquiéter de la réception d'une réponse par sondage ("polling") ou par un appel bloquant s'il souhaite que le processus requête/réponse soit synchrone. Ceci sera détaillé au sein du chapitre 5.

Ce besoin d'asynchronisme est illustré dans le document [9]. Les exemples donnés concernent le système de gestion d'objets distribués CORBA, mais collent bien à la situation du LONOS.

Portabilité: Le système LONOS doit être "le plus portable possible" et ceci en vue de l'implanter sur les plateformes suivantes: Windows 95/98, Windows NT et UNIX (Linux, Solaris, etc.). Ceci nécessite l'isolation de tous les modules dépendant du système d'exploitation tels que sont la gestion des communications (sockets TCP), la gestion des threads, la gestion des sémaphores et des mutex[1] (voir également [19] et [22]).

En outre, on pense à l'utilisation de classes Java pour implémenter certaines parties des serveurs. On pense aux parties qui seraient développées par les concepteurs de noeuds. Ainsi, plus besoin de développer du logiciel compilé pour différentes plateformes. On verra plus loin ce que ceci signifie exactement (cf. chapitre 3).

1. Ces notions seront rappelées dans une section ultérieure

Chapitre 3

La gestion des objets

3.1 Introduction

Ce chapitre présente la solution au besoin de gestion orientée-objets mis en évidence au chapitre 2. Le chapitre est axé sur trois parties principales: le modèle de l'orienté-objet en général, le modèle des objets du LONOS adapté à la gestion de points de contrôle et enfin quelques détails d'implémentation.

3.2 Principes de la gestion orientée-objet

3.2.1 Motivations

Les concepts de "l'orienté-objet" sont populaires dans le domaine de la programmation avec comme objectifs la facilité de réutilisation, de maintenance et d'interconnexion des différentes parties d'un programme. Plus récemment, les concepteurs de bases de données ont pu apprécier les avantages de l'orientation objet (notamment dans le cas où les données ne s'organisent pas simplement sous forme de lignes et de colonnes). La programmation orientée-objet et la gestion de bases de données orientées-objet sont des choses différentes mais qui partagent le même principe clé: l'encapsulation.

Les principes "orienté-objet" utilisés dans le modèle du LONOS sont plus proches de la gestion d'une base de données. L'intérêt réside dans la possibilité d'organisation des objets offerte par ces concepts.

3.2.2 Concepts de l'orienté-objet

Le concept central est l'objet. Un objet représente une entité du monde réel, ou du monde virtuel pour les objets immatériels, qui se caractérise par une identité, des états significatifs et un comportement. Un objet, en informatique, est une unité logicielle qui

contient une série d'états et d'opérations qui affectent ces états. L'objet représente donc un sous-sensemble des états et du comportement de l'entité représentée. On dit que ces états et opérations sont encapsulés au sein de l'objet.

Ces états et opérations ne sont pas forcément tous visibles (c.-à-d. manipulables de l'extérieur de l'objet). L'objet possède une *interface* qui publie un certain nombre d'états et d'opérations de l'objet.

Structure d'un objet

On désigne les états et les opérations d'un objet par les termes variables et méthodes respectivement. Ainsi, tout ce qu'un objet "sait" est contenu dans ses variables et tout ce dont il est capable est contenu dans ses méthodes.

Classe d'objets

Une classe est l'abstraction d'un ensemble d'objets qui possèdent la même structure (ensemble des caractéristiques) et un même comportement (ensemble des méthodes). On dit qu'un objet est une instance d'une et une seule classe.

En général, il y a un certain nombre d'objets représentant des entités du même type. Une classe d'objets représente donc un type d'entité et les instances de cette classe représentent les entités de ce type.

Héritage

Le concept de classe d'objets est puissant car il permet la création d'une multitude d'instances avec un minimum d'efforts. Ce concept est rendu encore plus efficace avec le mécanisme d'héritage.

L'héritage permet la définition d'une nouvelle classe d'objets en termes d'une ou plusieurs classes existantes. La nouvelle classe ainsi créée est appelée sous-classe et elle inclut automatiquement les méthodes et les variables des classes dont elle descend. Ces dernières sont nommées super-classes. Une sous-classe peut différer de ses super-classes à plusieurs niveaux:

1. La sous-classe peut contenir des méthodes supplémentaires.
2. La sous-classe peut surcharger la définition de n'importe quelle méthode ou variable d'une de ses super-classes en utilisant le même nom avec une nouvelle définition. Ceci fournit un moyen simple et efficace de manipuler certains cas.
3. La sous-classe peut restreindre une méthode ou une variable d'une de ses super-classes d'une certaine façon.

Le mécanisme d'héritage est bien entendu récursif. Ainsi, une sous-classe peut être super-classe de ses propres sous-classes. On construit alors une hiérarchie d'héritage.

3.3 Le modèle du LonOS

La gestion des objets au sein du LonOS est conçue dans le but de représenter des ressources physiques (monde réel) telles que des noeuds ou logiques (monde virtuel) telles que des connexions de variables. Un serveur sera responsable de la gestion des objets et du lien qui existe entre ceux-ci et les ressources physiques ou logiques (pour les noeuds et les connexions de variables, il s'agit du gestionnaire de réseau dont on a parlé au chapitre 2).

Afin de présenter le modèle de gestion d'objets du LonOS, effectuons une analogie (voir figure 3.1) entre l'exploitation de réseaux de points de contrôle et l'administration de systèmes vue par OSI (voir 1.3.1).

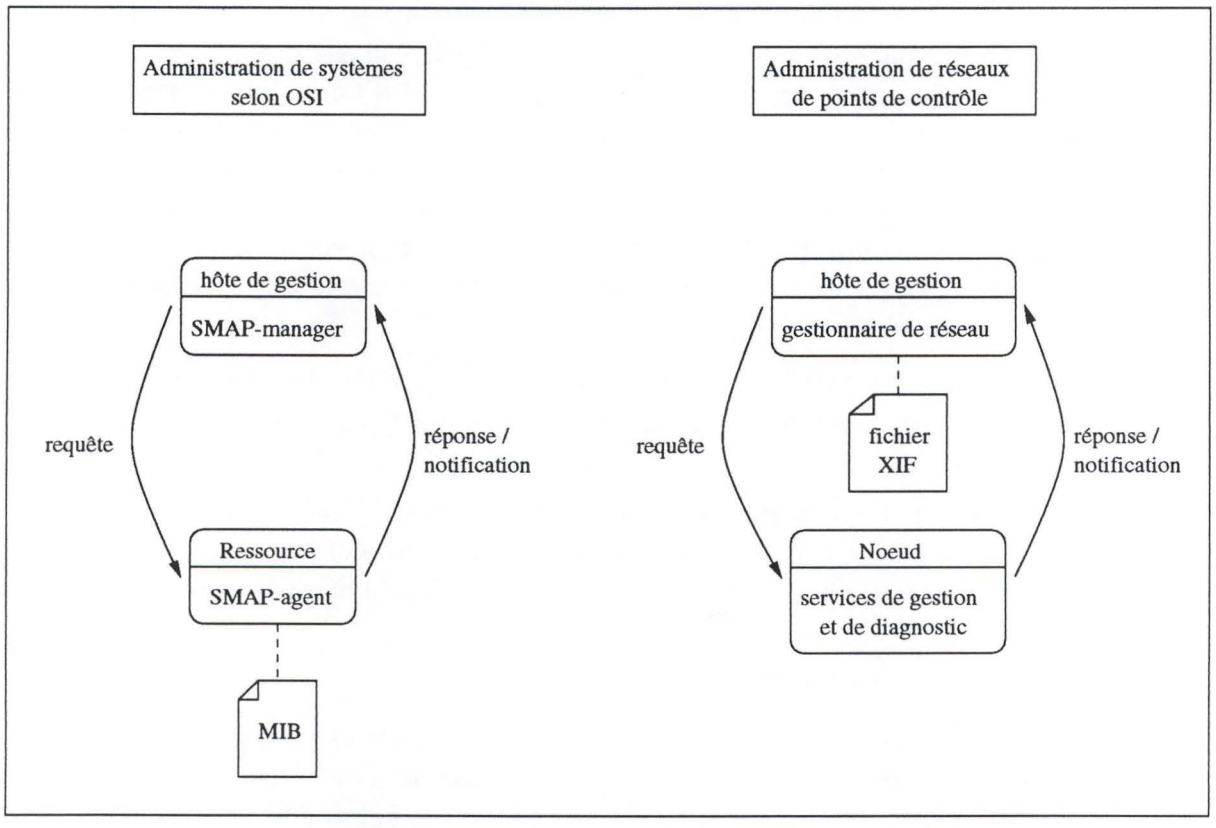

FIG. 3.1 – *Analogie avec l'administration de systèmes selon OSI.*

Dans le modèle d'OSI, l'administration s'effectue à partir d'hôtes de gestion sur lesquels sont exécutés des SMAP-managers. Ces derniers s'adressent alors à des SMAP-agents qui fonctionnent sur des hôtes (éventuellement les mêmes que pour les managers) ou sur les ressources elles-mêmes (dans le cas de certains routeurs ou switches, par

exemple). Ces agents publient une structure d'information appelée MIB décrivant les ressources gérées.

Dans le cas de l'exploitation de réseaux de contrôle, et en particulier dans le cas de la technologie LonWorks, le rôle d'agent est joué par les couches 6 et 7 du protocole LonTalk. C'est en effet par le biais de ces couches logicielles que peuvent être effectuées une série d'opérations de gestion et de diagnostic des noeuds du réseau (cf. 1.2). Pour rappel, il était possible de mener des actions telles que: l'interrogation de l'état d'un noeud, la configuration des adresses, l'installation d'une application. Les couches 6 et 7 du protocole LonTalk offrent donc une interface uniforme.

Cependant, en général, les noeuds diffèrent au niveau de leurs fonctionnalités (pensons à l'exemple d'application de la section 1.1.3 au chapitre 1), c.-à-d. au niveau de la sémantique de leurs variables-réseau. On veut dire par là qu'une variable-réseau d'un type donné va pouvoir être utilisée pour contrôler l'éclairement diffusé par une lampe, pour activer un photocopieur, pour contenir l'identifiant d'une carte magnétique, etc. Les couches 6 et 7, qui manipulent les messages de propagation de variables-réseau, n'ont aucune connaissance des ces différentes fonctionnalités. Seul le développeur peut fournir une information complémentaire. Ce que l'on recherche donc dans ce modèle objet, c'est pouvoir rendre compte de ces fonctionnalités en enrichissant les fichiers XIF (qui décrivent les variables-réseau en terme de type et d'index — voir 1.2) avec des méthodes propres au noeud géré (voir figure 3.2).

Cet enrichissement serait fourni par le concepteur du noeud sous la forme d'une classe d'objets qui permettra donc de spécialiser la représentation que l'on aura du noeud qu'il a construit.

Le modèle de gestion orientée-objet du LonOS repose donc sur deux concepts importants qui seront l'abstraction par objet et la spécialisation. Ces deux concepts seront matérialisés par les notions d'*ObjectClasses* et d'*ObjectInstances* décrites ci-après.

3.3.1 Notion d'ObjectClass

Une *ObjectClass* est une classe d'objets dans le modèle objet du LonOS. Elle est donc l'abstraction d'un ensemble d'objets qui ont la même structure et le même comportement. Ces objets représentant des ressources telles que des noeuds, des connexions de variables-réseau (voir la section 1.2). A cette effet, une *ObjectClass* diffère d'une classe d'objets habituelle par le fait qu'elle dispose de variables et de méthodes spécialisées. Celles-ci seront décrites plus loin.

Nous verrons qu'une *ObjectClass* possède une interface décrite sous forme d'un texte. Celle-ci spécifie ses variables et ses méthodes dans une syntaxe qui sera présentée plus bas. Cette interface cache l'implémentation de l'*ObjectClass* qui consiste principalement en une liste de procédures. Ces procédures sont implémentées en *C++* et placées dans

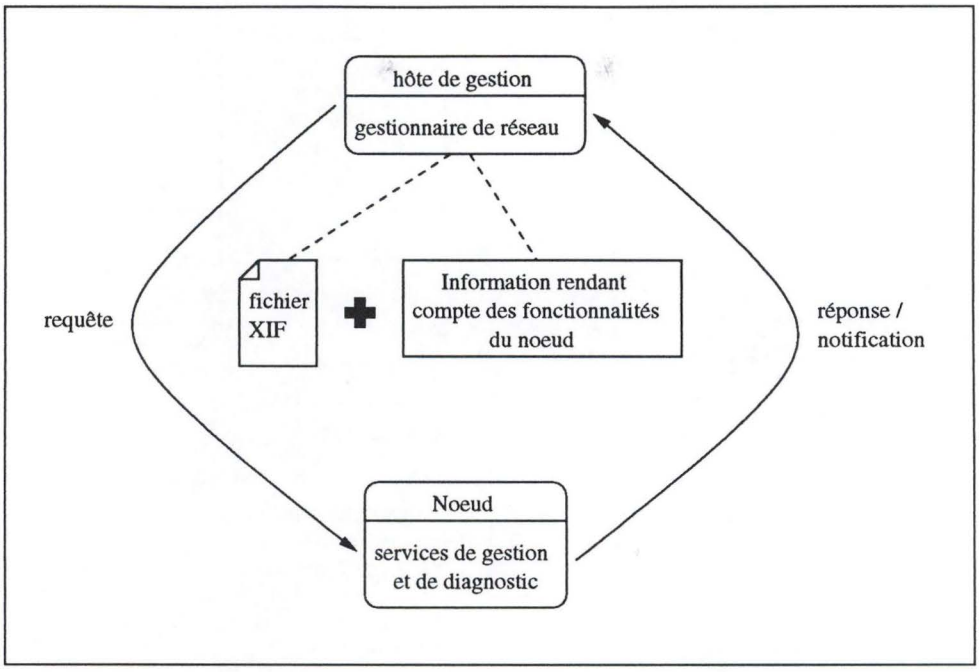

FIG. 3.2 – *Enrichissement de l'information concernant un noeud.*

des librairies chargées dynamiquement (voir les sections 3.5 et 5.2). Dans la suite du document, on confondra l'*ObjectClass* et son interface. Notons encore qu'une *ObjectClass* possède un nom qui l'identifie.

Une *ObjectClass* peut être définie sur base d'une autre *ObjectClass* selon le principe d'héritage (illustré à la figure 3.3). Dans le modèle de gestion orientée-objets du LONOS, cet héritage sera restreint: une *ObjectClass* contient les variables et méthodes de sa super-classe pour autant qu'elle ne les redéfinisse pas. Les variables ou méthodes redéfinies cacheront celles de la super-classe.

Dans la suite du document, il faudra décrire une série d'*ObjectClasses*, c.-à-d. leurs interfaces. Pour ce faire, on utilisera la syntaxe suivante:

objectclass <*class-name*> [**extends** <*class-name*>] **{**
    (<*type-def*> <*attribute-name*> ;)*
    (<*type-def*> <*method-name*> (<*params-def*>);)*
}

Dans cette syntaxe, <*class-name*> désigne le nom (identifiant) de l'*ObjectClass*; <*type-def*> désigne un type de données; <*attribute-name*> est un nom d'attribut; <*method-name*> est un nom de méthode; <*params-def*> est construit comme

<*params-def*> ::= [<*type-def*> (, <*type-def*>)*]

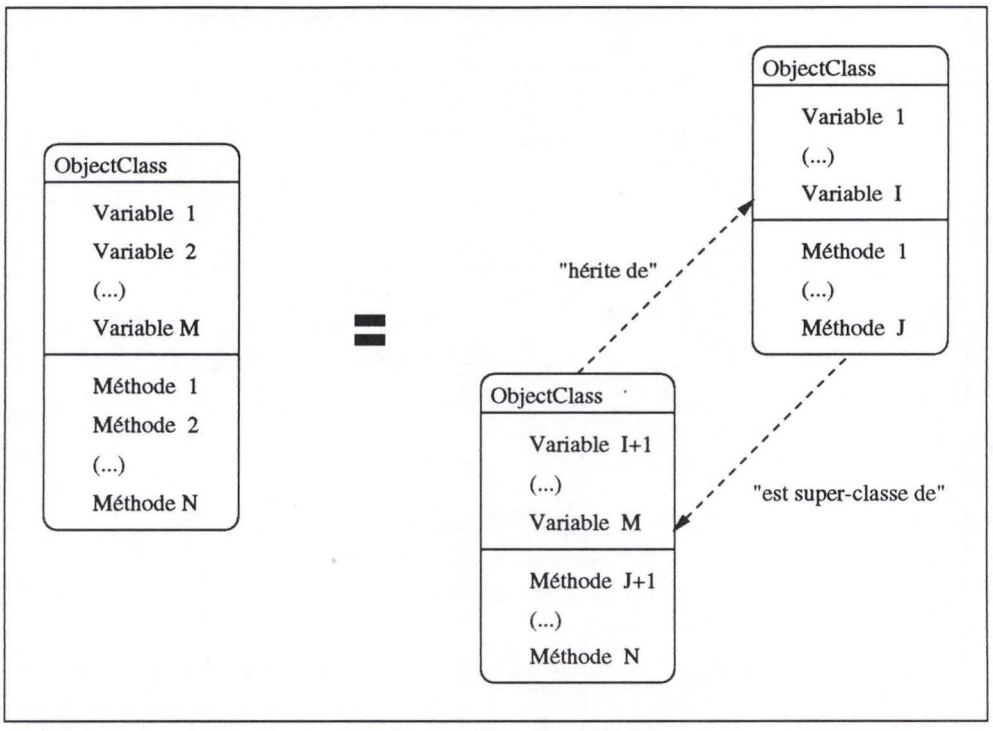

FIG. 3.3 – *Structure d'une ObjectClass et héritage.*

La section 3.3.5 donne un exemple d'abstraction d'un noeud LonWorks à l'aide du formalisme défini ci-dessus.

3.3.2 Notion d'ObjectInstance

Une *ObjectInstance* est une instance d'une *ObjectClass*, c.-à-d. qu'elle est un objet appartenant à l'ensemble abstrait par l'*ObjectClass*. Le fait qu'une *ObjectInstance* soit une instance d'une *ObjectClass* indique qu'elle représenté une entité réelle ou virtuelle dont le type correspond à cette *ObjectClass*.

Une *ObjectInstance* sera donc en fin de compte la représentation d'une ressource gérée par le système. Ceci est illustré à la figure 3.4.

3.3.3 Les attributs

Les variables décrites dans une *ObjectClass* seront nommées *attributs*. Comme dans le modèle objet classique, chaque attribut a un type. Les types supportés dans le modèle LONOS sont simples et composés. Les types simples sont les suivants (`<simple-type>`):

- Entier (long ou court): `integer` ou `long integer`,

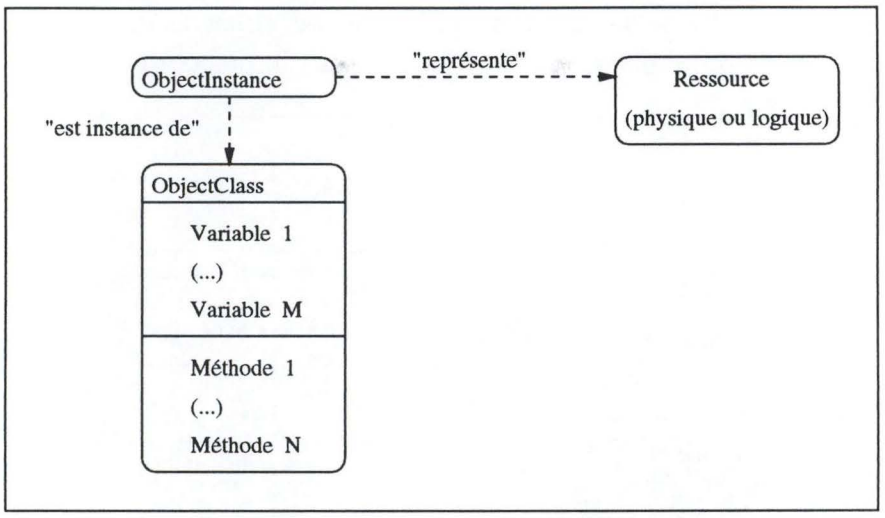

FIG. 3.4 – *ObjectInstance.*

- Caractère: `char`,
- Octet: `byte`,
- Booléen: `boolean`,
- Flottant: `float`,
- ObjectInstance (il s'agira en fait de la référence à une *ObjectInstance*): `object <class-name>`. La définition `<class-name>` renseigne le type de l'*ObjectInstance*, c.-à-d. sont *ObjectClass.*

Les types composés sont des tableaux (au sens du langage C) de types simples ou même composés. Ils sont représentés par

<*array*> ::= <*simple-type*> [<*size*>]

où <*size*> est la taille du tableau. Ainsi, en définitive, un type d'attribut est donné par:

<*type-def*> ::= <*simple-type*> | <*array*>

On distingue deux types d'attributs selon la façon dont on y accède: les attributs stockés (*stored attributes*) et les attributs accédés (*accessed attributes*). Ces deux types sont décrits ci-dessous:

Attributs stockés

Les attributs stockés (*stored attributes*) sont des attributs dont la valeur est stockée dans l'*ObjectInstance*, c.-à-d. que l'*ObjectInstance* référence des zones de mémoire

qui contiennent les valeurs de ces attributs. Ceci est illustré à la figure 3.5. Ce type d'attributs ne diffère donc pas d'une variable au sens habituel du terme.

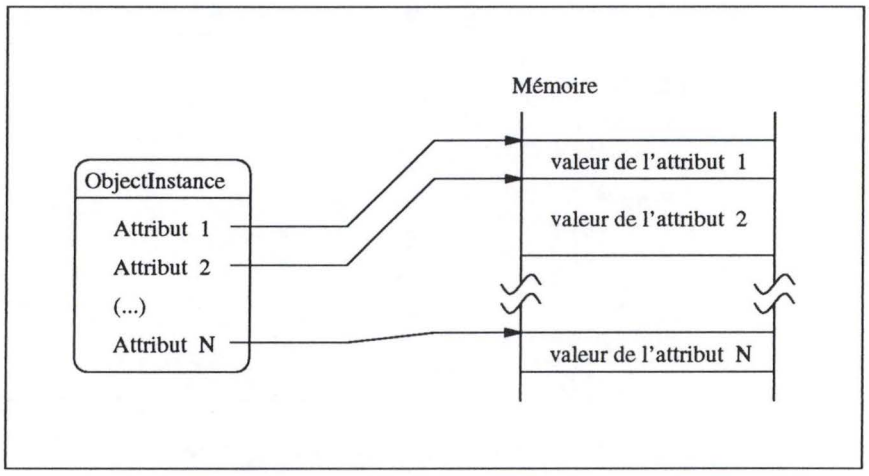

FIG. 3.5 – *Attributs stockés dans l'ObjectInstance.*

Attributs accédés

Les attributs accédés (*accesssed attributes*) ont des attributs dont la valeur n'est pas stockée dans l'*ObjectInstance*, mais au sein de la ressource représentée par celle-ci. On ne peut donc pas simplement aller lire en mémoire la valeur de l'attribut. Il faut s'adresser à la ressource représentée pour connaître la valeur de l'attribut ou pour modifier celle-ci. De plus, la façon dont on s'adresse à la ressource est intimement liée au type de cette ressource (un noeud LonWorks, une base de données, une connexion de variables nécessiteront des méthodes d'accès différentes).

Pour résoudre ce problème, on associe deux procédures à un attribut accédé (au niveau de la définition de l'*ObjectClass*, c.-à-d. au niveau de son interface). C'est par ces procédures et par un mécanisme décrit ci-dessous que l'on va accéder à la valeur de l'attribut. C'est pourquoi ces procédures seront appelées accesseurs (*accessors*) de l'attribut. Il existe donc une procédure pour récupérer la valeur de l'attribut, on la nommera accesseur *get()* et une procédure pour en modifier la valeur, appelée accesseur *set()*.

Lorsqu'on souhaite accéder à un "*accessed attribute*" d'une *ObjectInstance*, il faut déterminer quels sont ses accesseurs. Ceci est indiqué dans la définition de l'*ObjectClass* de l'*ObjectInstance*. Selon que l'on veut lire ou écrire la valeur de l'attribut, on utilise l'accesseur *get()* ou *set()* associé à cet attribut. Bien évidemment, tout ce mécanisme est transparent, c.-à-d. qu'il est exécuté au niveau de l'*ObjectInstance* sans qu'il ne soit

nécessaire de le spécifier lors de l'accès à l'attribut. On peut trouver une illustration du mécanisme des attributs accédés à la figure 3.6.

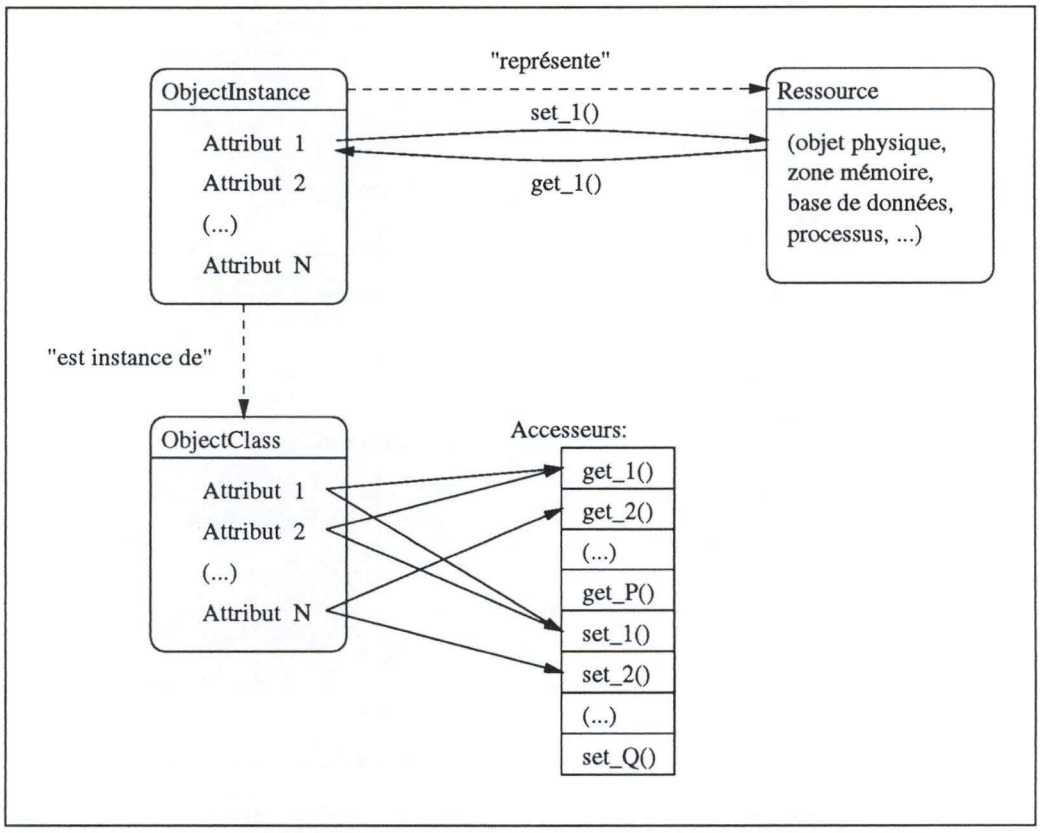

FIG. 3.6 – *Attributs accédés d'une ObjectInstance.*

Par exemple (voir figure 3.7), dans les objets représentant des noeuds, on peut trouver un certain nombre de variables-réseau. Les valeurs de celles-ci sont stockées dans le noeud physique et il est nécessaire d'effectuer une requête selon le protocole LonTalk afin d'y accéder. On peut donc définir des accesseurs *SNVT_get()* et *SNVT_set()* chargés d'effectuer ces requêtes. Pour un type de noeud particulier, on définira donc une ObjectClass comprenant la définition de ses variables-réseau sous la forme d'attributs accédés associés aux accesseurs *SNVT_get()* et *SNVT_set()*.

Il reste un point de détail à règler: il faut que l'on passe à un accesseur une référence à l'*ObjectInstance* qui contient l'attribut à accéder. En effet, il faut que l'accesseur sache à quelle ressource il doit s'adresser. Prenons l'exemple des ressources qui sont des noeuds LonWorks. Chacun de ces noeuds est représenté par une *ObjectInstance* qui a une *ObjectClass* comme type. Supposons que tous ces noeuds soient de même type

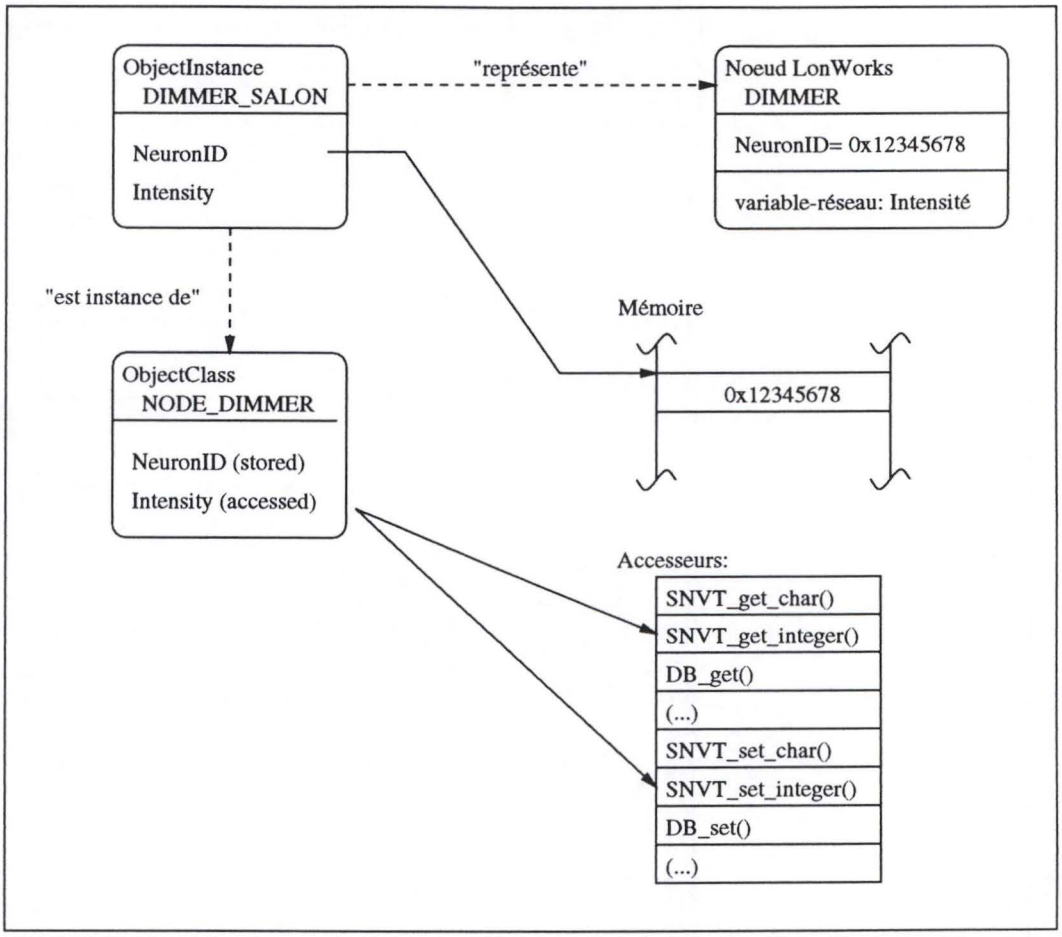

FIG. 3.7 – *Variables-réseau dans la représentation d'un noeud LonWorks.*

(ils auront donc la même *ObjectClass*) et qu'ils disposent d'une variable-réseau. Celle-ci est représentée comme un attribut accédé dans l'*ObjectClass*. Si on souhaite modifier la variable-réseau d'un noeud, on utilise son accesseur (*SNVT_ set()*). Mais comment celui-ci sait-il à quel noeud (physique) il doit s'adresser ? Il faut qu'il en connaisse l'identifiant, c.-à-d. son *NeuronID* (voir chapitre 1). Celui-ci devra être conservé dans l'*ObjectInstance* sous la forme d'un attribut stocké[1]. Ainsi, en passant la référence de l'*ObjectInstance* à l'accesseur, celui-ci peut s'adresser au noeud qu'elle représente.

Les accesseurs ont donc la forme suivante:

$$get : ObjectInstance \times String \Rightarrow Attr\_Type$$

1. Une autre méthode consiste à conserver une liste de couples composés du *NeuronID* et du nom de l'*ObjectInstance*.

$$set : ObjectInstance \times String \times Attr\_Type \Rightarrow \_$$

Il faut également prendre bonne note du point suivant: un attribut a un type indiqué dans l'interface de l'*ObjectClass*. Les accesseurs utilisés pour lire et écrire la valeur de cet attribut doivent donc correspondre à ce type.

La syntaxe utilisée pour représenter un attribut accédé permet bien entendu de spécifier les noms des accesseurs à utiliser pour lire et écrire la valeur de celui-ci. Cette syntaxe est donc:

<type-def> <attribute-name> { *<get-accessor-name>* , *<set-accessor-name>* }

3.3.4 Les méthodes

Les méthodes sont utilisées pour effectuer un traitement local à l'objet (c.-à-d. qui ne modifie que ses attributs stockés), pour cacher une requête sur l'entité représentée par cet objet (une requête qui modifiera l'état de l'entité représentée) ou pour toute combinaison de ces deux tâches. Ces méthodes sont définies dans l'interface de l'*ObjectClass* et cachent des procédures (un peu comme dans le cas des accesseurs).

Comme les méthodes peuvent avoir des paramètres variant en nombre et en types, il nous faut un mécanisme générique permettant le passage de n'importe quelle association de types simples et complexes. Le mécanisme repose sur une liste des types et des valeurs passées en paramètre (ce mécanisme sera décrit de façon plus détaillée au sein du chapitre 5). Ainsi, si on définit une méthode ayant n paramètres dont les types sont $Type1$ à $TypeN$, la procédure cachée par la méthode recevra une liste constituée de $\langle (Type1 : Value1), ..., (TypeN : ValueN) \rangle$.

En outre, les procédures cachées par les méthodes peuvent accéder aux membres (attributs stockés, attributs accédés et autres méthodes) de l'*ObjectInstance* dans laquelle ces dernières sont définies[2]. Ceci implique que les procédures doivent recevoir une référence à cette instance (tout comme les accesseurs). Une procédure a donc la forme suivante:

$$Procedure : ObjectInstance \times List \Rightarrow Return\_Type$$

La syntaxe de définition d'une méthode nécessite quelques commentaires. Premièrement, on peut spécifier dans cette définition quel est le nom de la procédure cachée par le méthode. Dans le cas où un nom n'est pas fourni, c'est le nom de la méthode elle-même qui sera utilisé. Ensuite, dans la signature d'une méthode, il n'est pas nécessaire de spécifier la taille des tableaux (types composés). En effet, cette taille sera déductible dynamiquement (voir le chapitre 5). Finalement, la syntaxe à utiliser est la suivante:

2. Il s'agit d'un abus de langage car ces méthodes sont définies dans l'*ObjectClass* dont l'ObjectInstance est l'instance.

<type-def> *<method-name>* (*<type-def>* (, *<type-def>*)*)
[{ *<procedure-name>* }] ;

Voici un exemple de l'intérêt de telles méthodes. Par exemple, on peut disposer, sur le réseau LonWorks géré par le système LONOS, d'un noeud capable de modifier la luminosité d'une lampe halogène (ce qu'on appelle communément un "dimmer"). A cet effet, le noeud dispose d'une variable réseau de type "entier" qui détermine la quantité d'éclairement requise. Une valeur de 255 signifie éclairement maximum tandis qu'une valeur de 0 signifie une lampe éteinte. Pour des raisons de facilité de gestion, le concepteur du noeud a fournit dans son ObjectClass de gestion du noeud "dimmer" des méthodes *SwitchON*, *SwitchOFF* et *SetIntensity* destinée à éteindre la lampe ou à l'allumer en la positionnant à la dernière valeur d'éclairement sélectionnée. Cette méthode gère une variable locale à l'objet et manipule la variable-réseau déterminant l'éclairement.

3.3.5 Application de la représentation objet

Prenons comme exemple d'application la représentation d'un noeud LonWorks à l'aide du modèle décrit ci-dessus. Il nous faut décrire l'objet de base à l'aide d'une ObjectClass:

```
objectclass NODE {
  integer NeuronID;
  void Reset();
  (...)
  void SNVT_set_integer(char [], integer);
  integer SNVT_get_integer(char []);
  void SNVT_set_string(char [], char []);
  char [] SNVT_get_string(char []);
  (...)
}
```

Le champ *NeuronID* correspond à l'identifiant du noeud physique représenté. Cette information est d'importance capitale car elle est à la base de la relation entre le noeud physique et l'instance d'objet 'NODE'. La méthode *Reset* permet d'effectuer une remise à zéro du noeud physique (par l'émission d'un message selon le protocole LonTalk). Les méthodes *SNVT_set_type* et *SNVT_get_type* sont utilisées pour accéder à une variable-réseau du noeud. Ces méthodes prennent en paramètre l'identifiant de l'attribut (char[]). Ces méthodes sont redondantes avec les accesseurs *SNVT_set* et *SNVT_get* dont on a parlé ci-dessus.

Remarquons que le champ *NeuronID* est un attribut stocké car le lien entre l'objet de type 'NODE' et le noeud physique doit être stocké au niveau du système LONOS. Il ne s'agit pas d'une information que l'on peut aller chercher "quelque part" ...

Cette classe d'objets permet d'accéder au noeud selon le protocole LonTalk. On a vu au début du chapitre que l'on souhaitait ajouter de l'information permettant de préciser les fonctionnalités d'un noeud particulier, c.-à-d. ce que le noeud est réellement capable de faire en dehors de la communication avec d'autres noeuds. La solution proposée est la définition d'une nouvelle ObjectClass correspondant au noeud que l'on veut représenter. Cette ObjectClass descendrait tout naturellement de la classe 'NODE' afin de bénéficier de ses fonctions de base.

Reprenons comme exemple la représentation d'un noeud permettant le contrôle d'une lampe, c.-à-d. le réglage de l'intensité lumineuse qu'elle doit délivrer.

```
objectclass NODE_DIMMER extends NODE {
  integer Intensity { SNVT_get_integer , SNVT_set_integer };
  void SwitchON();
  void SwitchOFF();
  void SetIntensity(integer);
}
```

Le champ *Intensity* représente une variable-réseau du noeud. Il s'agit ici d'un attribut accédé. Les procédures utilisées pour y accéder sont les procédures *SNVT_get_integer* et *SNVT_set_integer*.

Les méthodes *SwitchON* et *SwitchOFF* sont utilisées pour positionner l'intensité à 0 ou au maximum, respectivement. Utiliser la méthode *SetIntensity* est équivalent à effectuer un *Set* sur l'attribut *Intensity*.

3.4 Modèle des messages

Les interactions entre objets s'effectueront par messages. Cette section présente le modèle sous-jacent inspiré des JavaBeans[3] (voir [11]) et étendu à un environnement distribué (le framework LonOS).

3.4.1 Communication entre objets

Le paradigme retenu est donc une communication par messages. Supposons qu'un object A souhaite recevoir les messages d'un objet B (on dira que A souhaite être *listener* de l'objet B). L'objet A indique à une entité tierce qu'il souhaite recevoir les messages de l'objet B. La configuration est donc celle de la figure 3.8.

Selon son implémentation, l'objet B émet des messages à certain moments. Par exemple, à chaque modification de l'une de ses variables (par *set*()), un message est envoyé vers l'entité tierce (figure 3.9).

3. Il s'agit d'une recommandation pour la conception de classes Java interopérables.

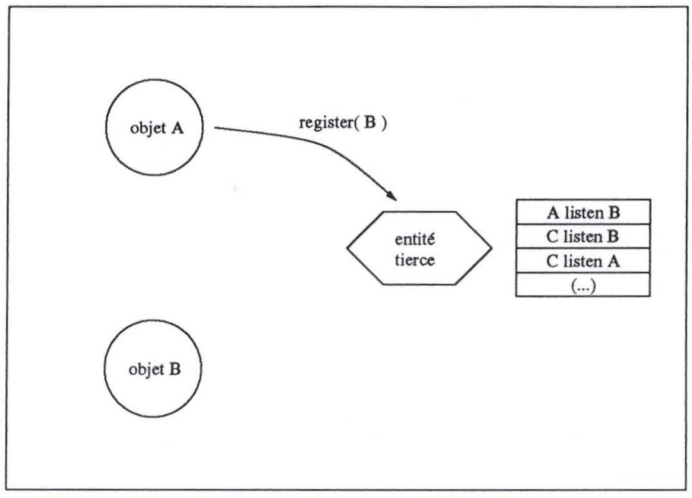

FIG. 3.8 – *L'objet A souhaite recevoir les messages de l'objet B.*

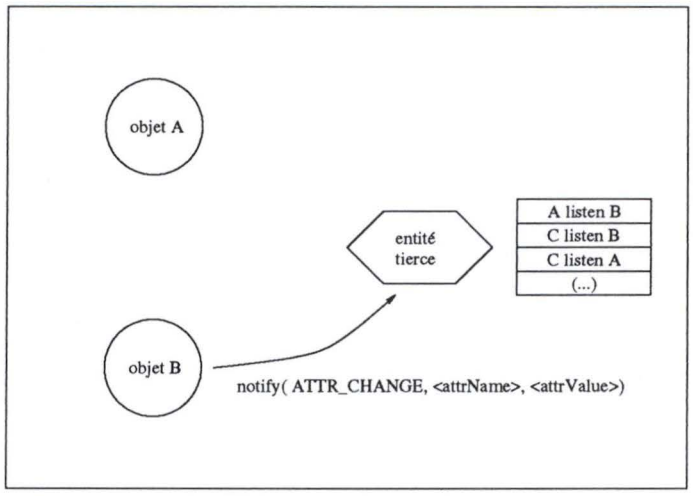

FIG. 3.9 – *L'objet B émet un message (ici un changement de valeur d'attribut).*

L'entité tierce qui reçoit les notifications et qui connait "qui écoute qui" est appelée "local multicaster". Cette entité est chargée pour chaque message qu'elle reçoit d'en envoyer une copie vers tous les objets qui se sont enregistrés comme *listener* de l'objet émetteur (figure 3.10).

Cette architecture nécessite l'apport de quelques modifications aux objets que l'on a décrit dans la section précédente:

- Au niveau de l'objet-émetteur B, il faut que la méthode *set*() de modification d'un attribut soit associée à l'émission d'un message de notification de cette mo-

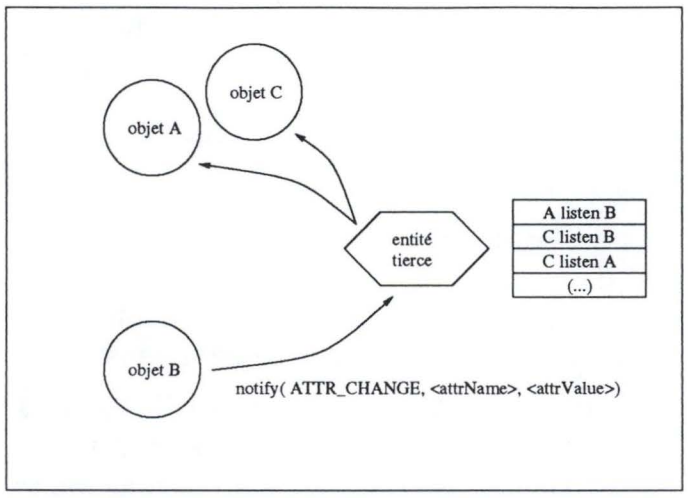

FIG. 3.10 – *Le "multicaster" envoie une copie de la notification aux objets intéressés.*

dification. De même, pour chaque cas où un message (d'un autre type ...) doit être envoyé, il faudra prévoir l'envoi vers le "multicaster". On peut envisager que tous les objets qui veulent implémenter la notification de modification d'attribut descendent d'une classe de base dans laquelle ce "job" est défini.

- Au niveau de l'objet "listener" A, il faut permettre la réception et le traitement des messages. Une méthode réservée $ProcessMsg()$ sera chargée de cette tâche. Le traitement d'un message relève donc de l'objet listener. Ce qu'il en fait dépend de son implémentation particulière.
- Le "local multicaster" doit permettre aux objets de s'enregistrer comme "listener" ou d'abandonner cette écoute ($RegisterListener()$ et $UnregisterListener()$). Il doit également recevoir les messages des objets émetteurs.

3.4.2 Communication entre objets distants

Ce point décrit la communication entre objets distants. Celle-ci repose sur une structure qui sera décrite au chapitre suivant, le framework. On considère dans ce point que le framework est une entité logicielle capable de gérer et de transmettre des messages entre d'autres entités logicielles qui lui sont connectées, les agents. En particulier, un agent peut contenir un gestionnaire d'objets fonctionnant selon le modèle décrit dans ce chapitre. Jusqu'ici, on a considéré uniquement la communication entre des objets situés sur un même agent. Dans le cas où les objets sont situés sur des agents distincts, les choses se compliquent. Prenons comme situation un système composé de deux agents qui gèrent chacun un objet: L'agent 1 détient l'objet A qui souhaite recevoir les notifications de l'objet B détenu par l'agent 2.

Comme dans le modèle précédent, chaque agent possède son multicaster (d'où la dénomination "local multicaster" — multicaster propre à un agent). L'objet A s'enregistre donc comme listener de l'objet B auprès du multicaster local, c.-à-d. celui de l'agent 1. Comme ce multicaster ne connait pas l'objet B, il s'enregistre auprès d'une entité située sur le framework et que nous nommerons "global multicaster" (quelle originalité ...).

Le multicaster du framework détermine l'agent qui détient l'objet B et s'enregistre auprès du multicaster de celui-i, c.-à-d. le multicaster de l'agent 2. Ceci ressemble à l'enregistrement du modèle précédent, mais n'oublions pas que nous sommes maintenant en présence de communications entre les agents et le framework.

Le multicaster de l'agent 2 place donc dans sa table des listeners une entrée associant l'objet B au framework. Ainsi, si plus tard un autre objet distant (situé sur l'agent 1 ou sur un autre) s'enregistre comme listener de l'objet B, il ne faudra plus modifier cette table locale, mais la table du framework.

La communication entre objets distants est illustrée à la figure 3.11.

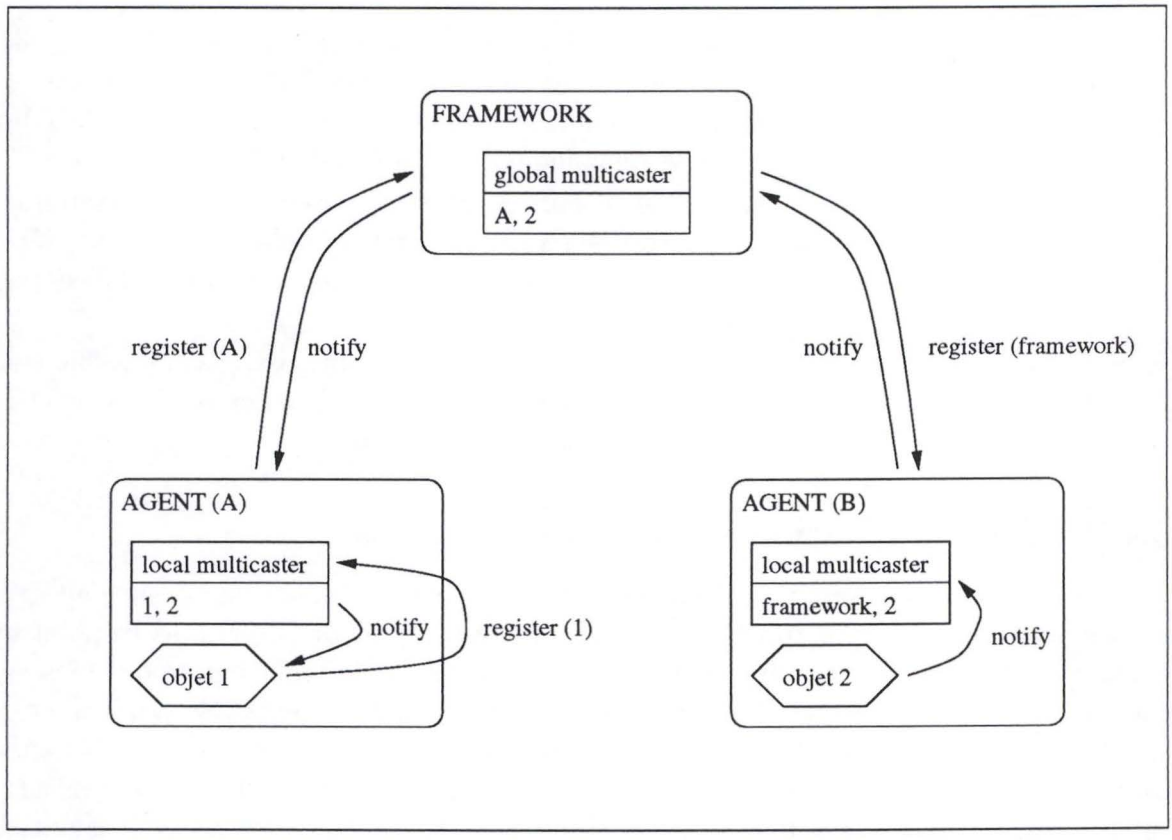

FIG. 3.11 – *Communication entre objets distants.*

3.4.3 Application de la communication entre objets

Maintenant que l'on a décrit un modèle et une architecture de communication entre objets, décrivons un exemple d'application. Celle-ci est issue du cas LonWorks. On a vue que le protocole LonTalk régissait l'échange de message relatif aux variables-réseau et qu'il était possible d'établir une "connexion" entre un ensemble de celles-ci.

Il peut être utile de représenter ces connexions par des objets. Mieux, on voudrait créer des connexions qui ne sont plus régies par le protocole LonTalk, mais qui sont gérées par le système LonOS. On ne peut pas connecter des variables-réseau de noeuds situés sur des réseaux distants. Des objets "connexion de variables" virtuels apportent alors une solution à ce problème. En effet, la gestion de la propagation des variables ne relèverait plus du protocole LonTalk au sein de ces connexions virtuelles, mais reposerait sur un échange de messages entre des objets représentant les noeuds dont les variables doivent être connectées.

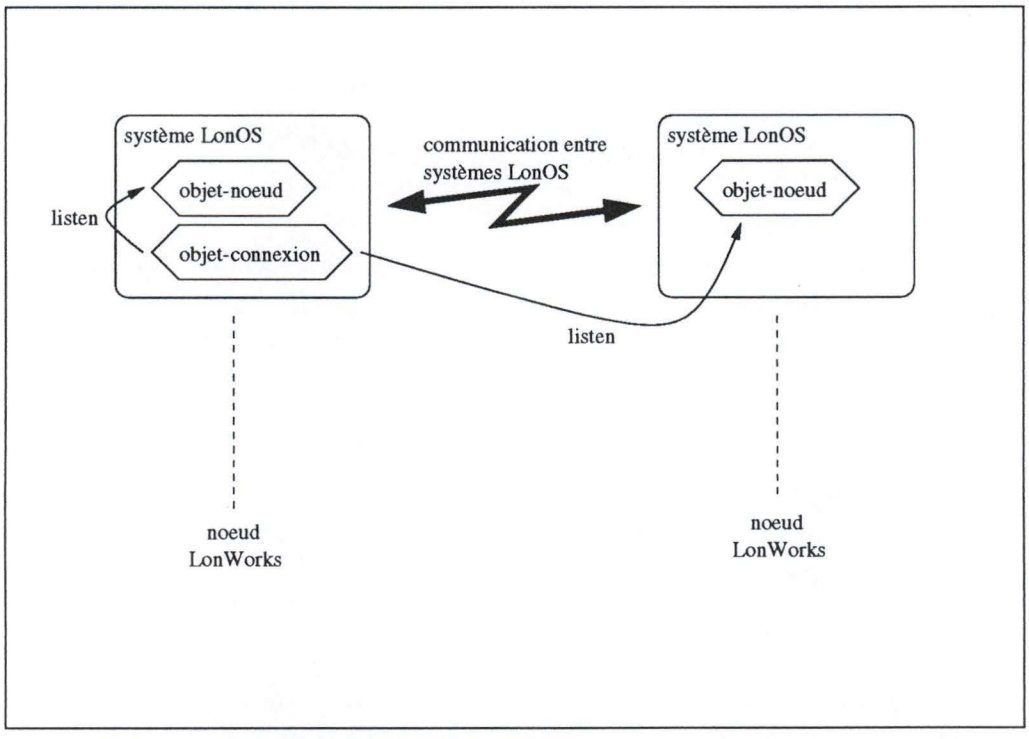

FIG. 3.12 – *Connexions virtuelles.*

Bien que sur la figure 3.12 qui illustre cette application les noeuds soient liés à des systèmes LonOS différents, on peut très bien avoir le cas de plusieurs réseaux LonWorks (ou autres) gérés par un même système LonOS (on est toujours dans le cas d'un échange de messages entre objets distants).

3.5 Implémentation

La gestion des objets est la responsabilité d'une entité logicielle appelée *ObjectManager* (voir figure 3.13). Cette entité connaît la description des *ObjectClasses* (leurs interfaces) ainsi que leur implémentation. L'*ObjectManager* gère également les instances des *ObjectClasses*, c.-à-d. les *ObjectInstances*.

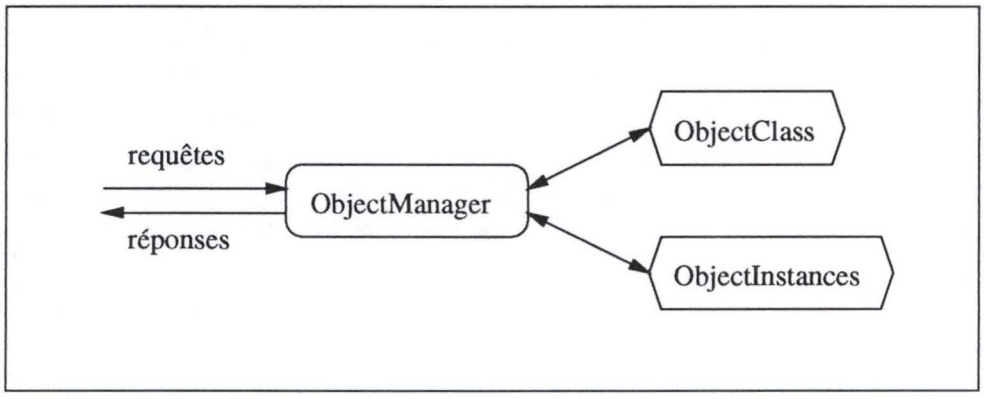

FIG. 3.13 – *L'ObjectManager.*

Ainsi, lorsqu'une application veut manipuler un objet (une *ObjectInstance*), elle s'adresse à l'*ObjectManager* à l'aide d'un jeu de requêtes (le chapitre 4 présentera plus en détail la structure de celles-ci). La manipulation d'un objet comprend les opérations suivantes:

- Créer un objet, c.-à-d. instancier une *ObjectClass*;
- Détruire un objet;
- Changer la valeur d'un attribut d'un objet;
- Lire la valeur d'un attribut d'un objet;
- Appeler une méthode d'un objet.

On peut, en sus, imaginer une opération permettant de déterminer l'interface de l'*ObjectClass* dont un objet est l'instance. Une application aurait donc l'opportunité d'apprendre la description d'un objet dont elle ne connaît que le nom.

Les interfaces des *ObjetClasses* sont conservées dans une table de *hash*. Il en est de même pour les *ObjectInstances*. Ceci permet à l'*ObjectManager* de retrouver rapidement la classe ou l'instance spécifiée dans une requête qui lui est adressée.

L'*ObjectManager* possède également les implémentations des *ObjectClasses*, c.-à-d. les procédures cachées derrière les accesseurs et les méthodes (illustration à la figure

3.14). Ces procédures peuvent être définies de trois façons. Premièrement, elles font partie du code de l'*ObjectManager*. Deuxièmement, elles sont définies au sein de librairies que l'on peut charger dynamiquement (*shared libraries* sous UNIX, DLL sous Windows). Et enfin, elles peuvent être définies dans des classes Java dont les méthodes sont appelées via l'interface JNI[4] (Java Native Invokation, cf. [10]). Cette dernière possibilité répond au besoin de portabilité dont on a déjà fait mention au chapitre 2. Le développeur d'une *ObjectClass* ne doit donc développer qu'une implémentation.

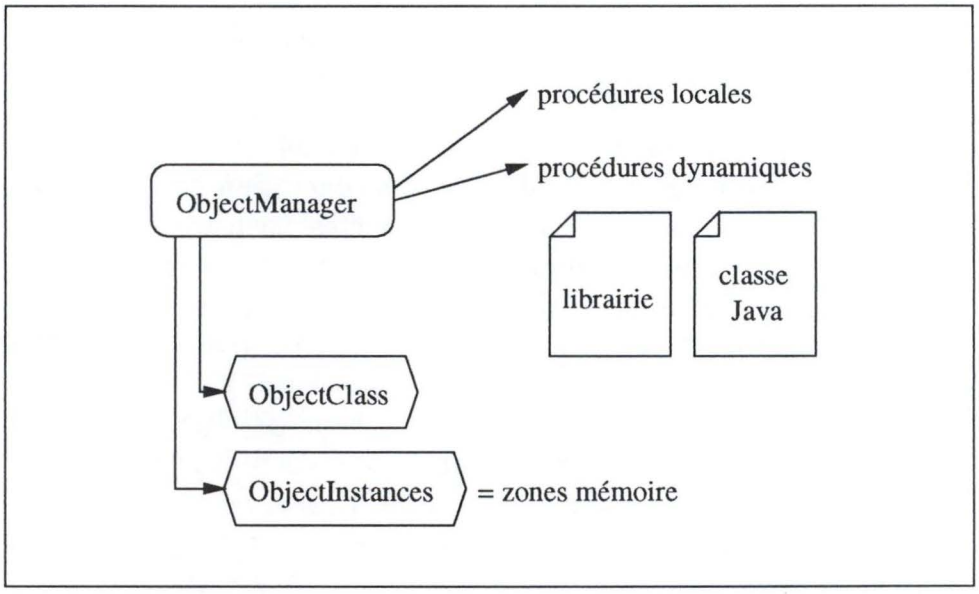

FIG. 3.14 – *Gestion des méthodes et accesseurs.*

3.6 Conclusion

Les avantages de l'utilisation d'une telle gestion sont les suivants:

1. La définition des *Objectclass* est indépendante de la plate-forme dans le cas où on utilise des classes Java. Il ne s'agit pas d'un langage propriétaire mais d'un langage répandu !
2. Une *ObjectClass* n'est pas nécessaire si on ne souhaite pas fournir d'information supplémentaire sur le noeud. Il existe en effet un fichier généré à la compilation du firmware d'un noeud LonWorks qui contient la définition des variables-réseau du

4. Le test de cette approche ne s'est pas avéré concluant dans une environnement multiprocessus avec la version actuelle du JDK (une erreur de segmentation a lieu au sein de la librairir JNI).

noeud (fichier XIF — eXternal Information File). Celui-ci est toujours nécessaire et peut être utilisé comme *ObjectClass* basique.

3. On peut aisément définir des noeuds virtuels qui possèdent de puissantes méthodes ainsi que la possibilité d'effectuer des affichages contrôlés par des événements réseau.
4. On peut spécialiser une définition d'objet pour permettre l'exploitation des fonctionnalités de l'entité représentée (un noeud, par exemple).

Pourquoi ne pas avoir utilisé un système de gestion d'objets tel que CORBA? Pour deux raisons principales:

1. CORBA n'est pas suffisamment asynchrone (cf. [9]). La solution fournie par le LonOS sera décrite dans le chapitre 4 — La conception du framework. CORBA fournit bien la possibilité d'effectuer des requêtes asynchrones (paradigme "*one-way*"), mais il s'agit d'une solution "best-effort" sans réponse.
2. Il est nécessaire de recompiler l'application lorsque l'on ajoute un type d'objet (*ObjectClass*). Ce qui n'est pas intéressant dans notre cas puisqu'on souhaite pouvoir ajouter un paquetage matériel et logiciel au système. CORBA offre la possibilité d'effectuer des opérations de manière statique sur les objets, par l'intermédiaire de stubs (comparables aux interfaces spécialisées décrites au chapitre 5) ainsi que des appels dynamiques, en récupérant la description de l'objet (son *ObjectClass* en quelque sorte), mais l'implémentation de ces objets est statique (définie en C, C++, ...).

Cependant, peut être qu'une version future de CORBA répondra aux besoins du LonOS, notamment avec une interface Java/JavaBeans ...

On peut encore apporter une amélioration à la gestion d'objets présentée dans ce chapitre. La syntaxe utilisée pour spécifier une *ObjectClass* n'est compatible avec aucune gestion d'objets connue. Il existe pourtant un langage de définition d'interfaces appelé IDL — Interface Definition Language — qui commence a être largement utilisé.

Maintenant que l'on a vu comment un serveur gérait des objets, il nous faut décrire la façon dont il va interagir avec ses clients et avec les autres serveurs. Le chapitre 4 décrit la structure de communication, le framework, chargée d'interconnecter les clients et serveurs.

Chapitre 4

La conception du framework

4.1 Introduction

Au chapitre 2, on a présenté quatre besoins qui sont développés dans ce document: une gestion orientée-objets que l'on vient de décrire au chapitre 3, un support pour de multiples clients et de multiples serveurs, un accès distant et ce que nous avons nommé l'anonymat. On apporte une solution à ces trois derniers besoins par une architecture de communication appelée *framework*. Celle-ci est présentée au sein de ce chapitre.

Au terme de ce chapitre, des serveurs particuliers seront brièvement présentés: le serveur *LonTalk* destiné à la gestion de réseaux LonTalk et le serveur *Remote* qui permet d'accéder à des objets distants, c.-à-d. géré par un serveur exécuté sur une autre machine.

4.2 Fonctionnalités

Le framework (voir la figure 4.1) est le processus de base du système LonOS, celui qui permet l'interconnexion des serveurs et des clients. Il permet premièrement de réduire le nombre de connexions entre les clients et les serveurs (on a pu voir, sur la figure 2.3, un exemple où chaque client était connecté à tous les serveurs, ce qui est la pire situation). Il doit être capable de gérer les divers serveurs, c.-à-d. lancer leur exécution, gérer leur connexion, tester leur présence, etc. Il doit également être en mesure de gérer les clients de ces serveurs et de router les requêtes des clients vers les serveurs concernés (en effet, on a vu au chapitre 2 qu'un client ne connaissait à priori pas le gestionnaire d'un objet).

L'architecture du framework est inspirée de la suite logicielle OpenMaster qui repose également sur une structure de communications centrale. Le framework d'OpenMaster est décrit dans le document [20].

La figure 4.2 montre un exemple d'une configuration typique. L'application est, ici, la gestion de noeuds LonWorks, mais le framework doit être générique afin de supporter

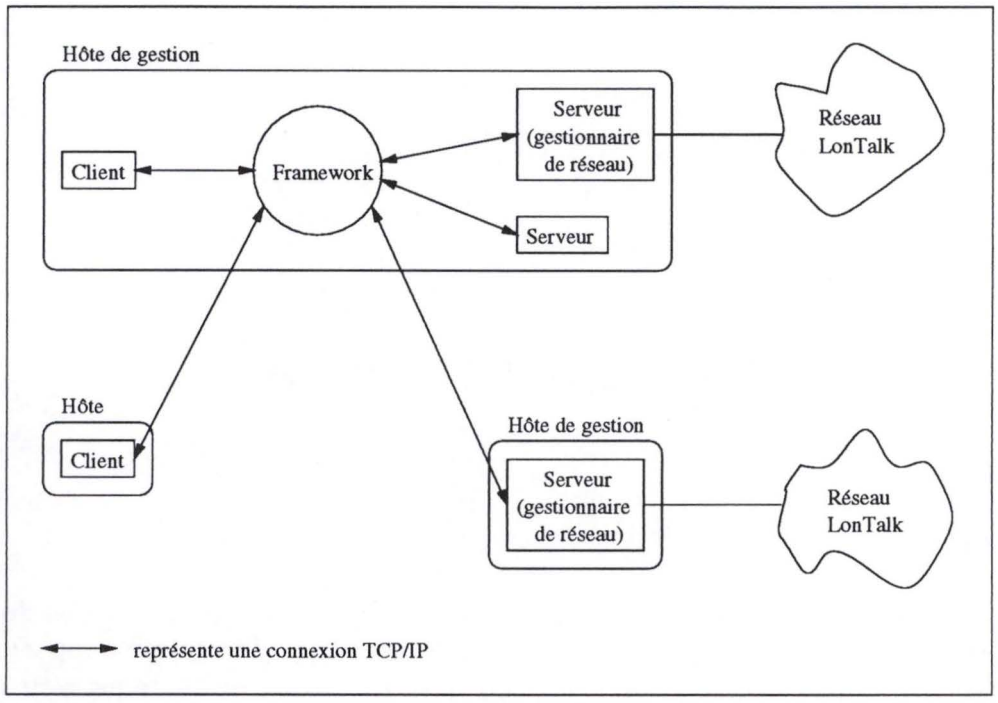

FIG. 4.1 – *La structure centrale qu'est le framework.*

n'importe quel type d'objets (une base de données, des périphériques, des serveurs de monitoring distants, etc.). Ainsi, à la figure 4.2, on peut observer la présence de trois types de serveurs: un serveur LonTalk (en fait, un gestionnaire de réseau LonTalk), un serveur distant (celui-ci sera présenté à la fin de ce chapitre) et un serveur de sécurité (c'est un exemple de serveur gérant une base de données d'utilisateurs). On remarque également que deux serveurs de type LonTalk sont exécutés, ceci probablement parce qu'il existe deux réseaux LonTalk connectés au système: l'un pour le système de chauffage et l'autre pour l'éclairage.

4.2.1 Gestion des serveurs

La gestion des serveurs comprend plusieurs volets: l'initialisation des serveurs, l'enregistrement des serveurs et leur contrôle.

Initialisation des serveurs

L'initialisation des serveurs a lieu lors du démarrage du framework et consiste à démarrer une série de serveurs prédéfinis et décrits dans un fichier de configuration. Cette initialisation est simplement la création des processus serveurs prédéfinis. Il reste

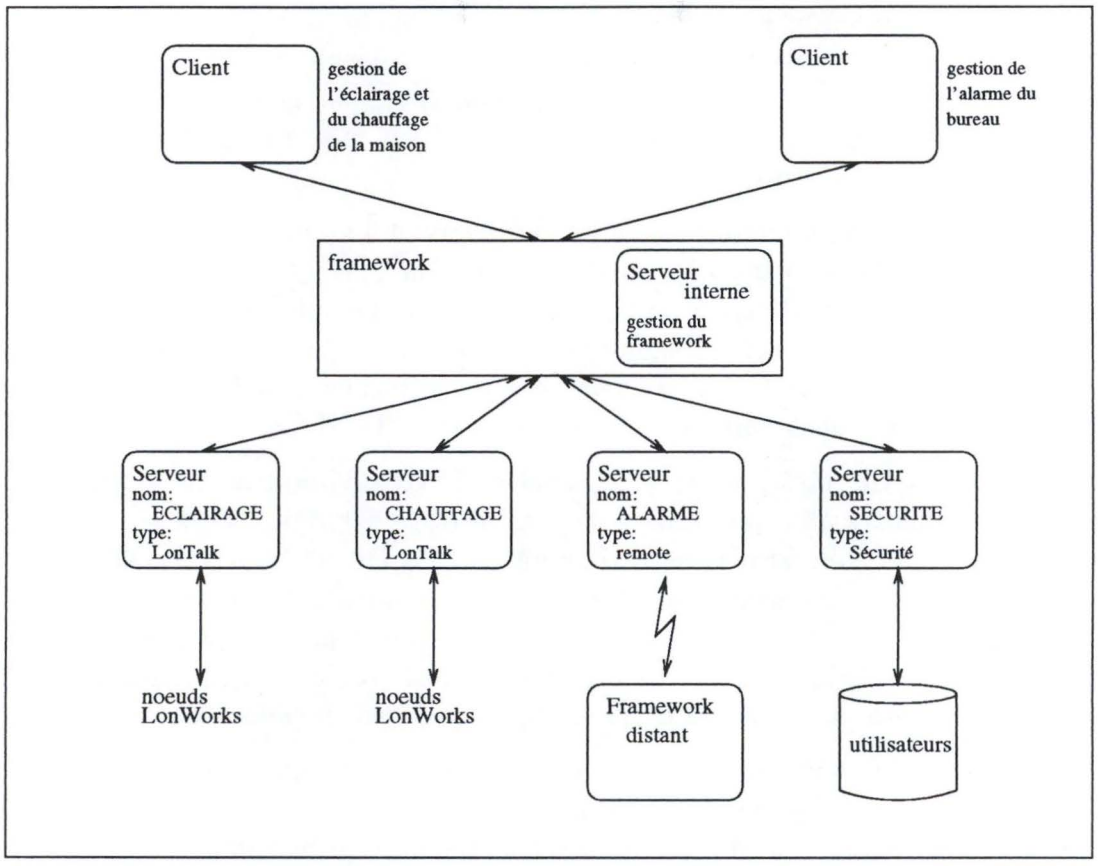

FIG. 4.2 – *Un exemple de configuration typique*

toujours possible de lancer un serveur supplémentaire après le démarrage du *framework*.

Par exemple, pour la configuration de la figure 4.2, les serveurs suivants on été démarrés.

- serveur LonTalk (ECLAIRAGE)
- serveur LonTalk (CHAUFFAGE)
- serveur Remote (ALARME)
- serveur Security

Enregistrement des serveurs

Lorsqu'un serveur a été démarré (par le framework ou de façon manuelle), il doit se connecter au framework, c.-à-d. établir un moyen de communication avec le processus framework afin de pouvoir recevoir les requêtes des clients et éventuellement afin de pouvoir émettre des requêtes vers d'autres serveurs.

Afin qu'un serveur puisse être reconnu par ses clients, il faut qu'il soit nommé au niveau du framework. Ainsi, lorsqu'un serveur s'est connecté au framework, il effectue une opération que l'on appelera "**Bind**" et qui consiste à passer son nom au framework. Ce nom doit bien entendu être unique au niveau d'un même framework. Dans le cas contraire, l'opération "**Bind**" échouera.

Cette opération de nommage est importante lors de l'ajout de nouveaux objets. En effet, cette opération ne peut être couverte par l'anonymat. Il est nécessaire d'indiquer sur quel serveur on souhaite créer un nouvel objet. Par exemple, si on a ajouté un noeud (physique) sur le réseau de chauffage, il faut ajouter un objet représentant ce noeud au sein du serveur 'CHAUFFAGE' afin que les clients puissent y accéder. Cet ajout ne peut se faire qu'en s'adressant explicitement[1] au serveur 'CHAUFFAGE'.

Une opération de retrait de serveur appelée "**UnBind**" doit avoir lieu avant qu'un serveur se déconnecte afin que celui-ci soit retiré de la table des serveurs connus. Si le serveur plante alors qu'il est enregistré, l'opération "**UnBind**" ne sera pas effectuée alors que le serveur n'est plus accessible. Il existe une première solution à ce problème. En effet, si un serveur plante et que sa connexion avec le framework est rompue, le framework peut le détecter[2] et enlever le serveur concerné de cette table. Dans les autres cas, cette détection incombera au processus de monitoring que nous dévoilerons plus loin.

Un problème de cohérence peut-il survenir suite aux ajouts et retraits de serveurs enregistrés? Le seul problème qui puisse se poser est qu'un client s'adresse à un serveur qui n'existe pas ou qui n'existe plus (le serveur peut s'être déconnecté pendant que le client demandait la liste des serveurs connectés au framework). Dans ce cas, le framework répondra qu'il ne connait pas le serveur demandé.

Contrôle des serveurs

Le contrôle des serveurs est une opération qui consiste à tester régulièrement si un serveur est toujours fonctionnel. En effet, il est possible qu'un serveur "tombe" durant le fonctionnement du framework. Dans ce cas, il faut que le framework soit capable de détecter l'incident et, si possible re-démarre le serveur concerné.

On a vu plus haut (voir Enregistrement des serveurs) qu'il était possible de détecter que la connexion avec le serveur était rompue. C'est donc un premier moyen de détection d'un malfonctionnement (il peut s'agir d'un processus qui s'est planté ou d'un processus qui a été tué par l'utilisateur).

1. Quoiqu'il soit possible de disposer à l'intérieur de chaque serveur de type LonTalk d'un système de "découverte" qui permettrait de connaître les noeuds du réseau qui ne sont pas représentés par un objet. On serait alors à même de se passer de l'identification explicite du serveur sur lequel on veut ajouter un objet. Cet ajout pourrait même se faire automatiquement.
2. Grâce à TCP (voir [16]).

Ce contrôle peut également être effectué par un système de *watchdog* (ou chien de garde) qui consiste à interroger régulièrement le serveur en lui envoyant une requête (par exemple "ARE YOU UP?") à laquelle il doit répondre (par exemple "YES") endéans un certain temps. Dans le cas contraire, le framework peut tuer le processus et le redémarrer.

Un autre moyen de détecter un malfonctionnement d'un serveur est de considérer sa file de message. On verra plus loin, lors de la présentation de l'architecure (section 4.3), que le framework possède une file de messages FIFO par serveur connecté dans laquelle viennent atterrir tous les messages qui leur sont destinés. Si cette file de message "déborde", cela peut être le signe d'un problème. Il peut également s'agir d'un serveur qui reçoit énormément de messages et qui n'a pas le temps de les traiter suffisamment rapidement (il faut peut être agrandir sa file de messages).

4.2.2 Routage des requêtes

Les requêtes des clients concernent des ensembles d'objets (au sens *ObjectInstance* du chapitre 3). Ces derniers sont gérés par des serveurs bien déterminés. Cependant, un client n'adresse jamais sa requête directement au serveur concerné (mise à part l'instanciation d'une *ObjectClass*). Il envoie sa requête vers le framework en indiquant vers quel objet ou groupe d'objets la requête est dirigée. C'est le rôle du framework de router cette requête vers le serveur auquel elle est réellement adressée. Il s'agit du concept d'anonymat dont on a déja parlé (pour rappel, voir chapitre 2 ou [9]).

Sur quelle base le framework va-t-il être capable de diriger une requête vers le "bon" serveur? Il faut qu'il sache quels objets chaque serveur gère. A cette fin, une fois qu'un serveur est connecté au framework (y compris le "**Bind**"), il doit indiquer la liste des objets qu'il gère. On appelera cette opération "**Enrol**" (vocabulaire relatif au framework OPENMASTER). La figure 4.3 illustre ces opérations lors du démarrage d'un serveur.

L'opération **Enrol** consiste donc à ajouter une information dans la table de routage du framework. Sur base du contenu de cette table, le framework sera capable de diriger une requête d'un client (ou d'un serveur agissant comme tel) vers le serveur qui a enregistré l'objet désigné dans la requête.

Il existe également une opération permettant de retirer un objet enregistré au niveau du framework. Celle-ci est nommée "**UnEnrol**". Elle est utilisée par un serveur lorsqu'une instance d'objet publiée (c.-à-d. que son nom a été transmis au framework) est supprimée. Ceci doit également avoir lieu préalablement à une déconnexion du serveur. Les remarques concernant l'opération "**UnBind**" sont aussi valable dans ce cas (un serveur qui est déconnecté sans avoir retiré les objets qu'il a publié).

Dans une requête, un client peut donc désigner un objet par son nom: `DIMMER_SALON`, ou un groupe d'objets en utilisant des masques: `DIMMER_*`.

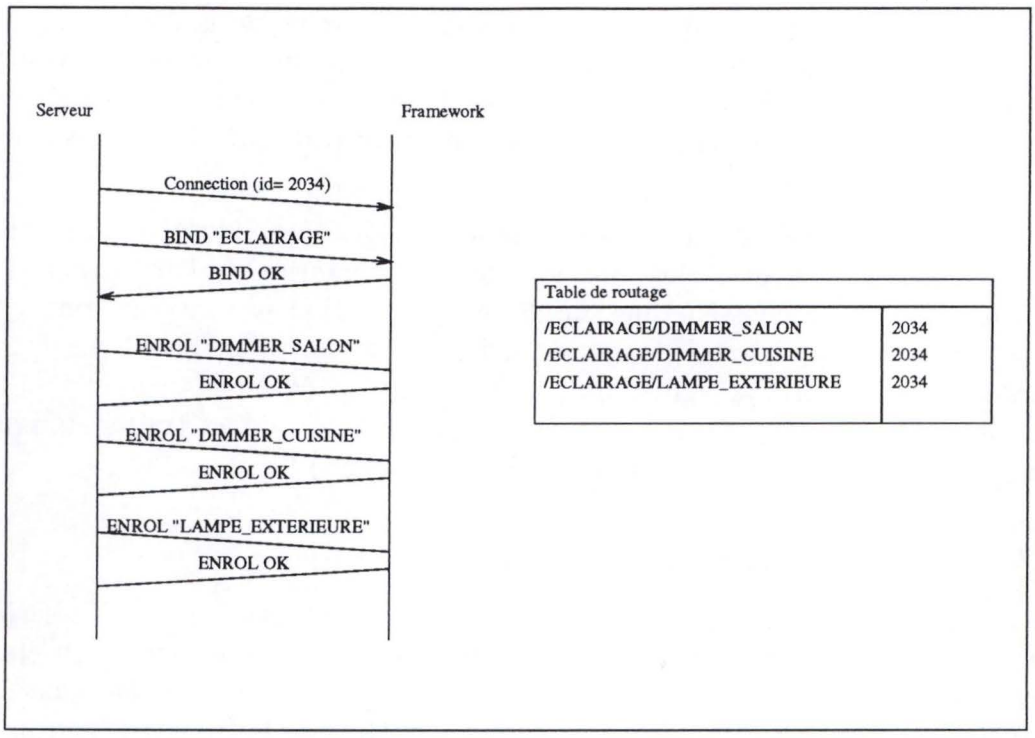

FIG. 4.3 – *Exemple de démarrage d'un serveur: "bind et enrols"*

Structure des messages

Quelle est la structure d'une requête? Quelles sont les informations qu'elle doit contenir afin qu'elle puisse transiter vers le serveur auquel elle est destinée et que celui-ci puisse envoyer une réponse?

Remarquons qu'il n'y a pas que des requêtes qui transitent par le framework. En effet, on a vu au chapitre 3 que des messages de notification pouvaient également être échangés. On va donc d'abord décrire la structure d'un message générique, unité de base transitant par le framework. Ce message pourra alors être spécialisé en requête, en notification ou en tout autre message spécialisé (en prévision d'extensions futures).

La structure d'un message est la suivante:

- Taille du message;
- Type du message (Requête, notification, etc.);
- Identification de l'émetteur;
- Identification du destinataire;
- Contenu du message.

Le champ *Taille du message* contient le taille totale du message (ceci est utilisé afin de recomposer un message segmenté par TCP, cf. [16]). Le champ *Type du message* permettra de distinguer les requêtes des notifications ou d'autres types de messages. Actuellement, on distinguera trois types de messages: REQUETE, REPONSE et NOTIFICATION. Les champs *Identification de l'émetteur* et *Identification du destinataire* sont utilisés par le framework pour diriger le message vers le destinataire requis et afin que ce dernier puisse émettre une réponse vers l'émetteur. Remarquons que le champ *Identification du destinataire* peut être "vide" dans le cas d'une requête puisque dans ce cas, le destinataire est déterminé sur base d'un nom d'objet enregistré. Enfin, le restant du message contient des informations diverses et dont la nature dépend du type de message.

Structure des requêtes

Une requête est un message dont le type est REQUETE et qui contient trois informations principales:

- Un nom d'opération;
- Un nom d'objet (ou un groupe d'objets);
- Une série de paramètres nécessaires à la bonne exécution de l'opération;
- Un identifiant de la requête.

Le nom d'opération désigne l'opération que le serveur devra exécuter sur l'objet identifié dans la requête. Par exemple, s'il s'agit d'une requête destinée à changer la valeur d'un attribut d'un objet, le nom d'opération sera 'SET' et les paramètres fournis seront le nom de l'attribut à modifier et la nouvelle valeur. D'autres opérations sont 'GET', 'CREATE' (qui permet de créer une instance d'un objet sur un serveur), 'DESTROY' (qui permet la suppression d'une instance), 'CALL' (qui permet l'appel d'une méthode d'une *ObjectInstance*), etc.

Comme les paramètres d'une opération peuvent être de types très divers, il a fallu mettre sur pied un système de sérialisation de ces paramètres. On convertit un ensemble de paramètres en une seule chaîne de caractères qui constituera le champ *Paramètres* de la requête (voir figure 4.4). Ce processus de sérialisation est inspiré de celui qui est utilisé pour la sauvegarde d'instances de classes Java. On peut se reporter à la référence [12] pour en apprendre plus.

Le champ *Identifiant de la requête* est un nombre qui est utilisé du côté client afin d'identifier la requête et la réponse qui y sera normalement associée. A l'avenir, cet identifiant pourra également être utilisé à des fins de gestion de transaction. Cette possibilité n'est cependant pas encore mise en oeuvre.

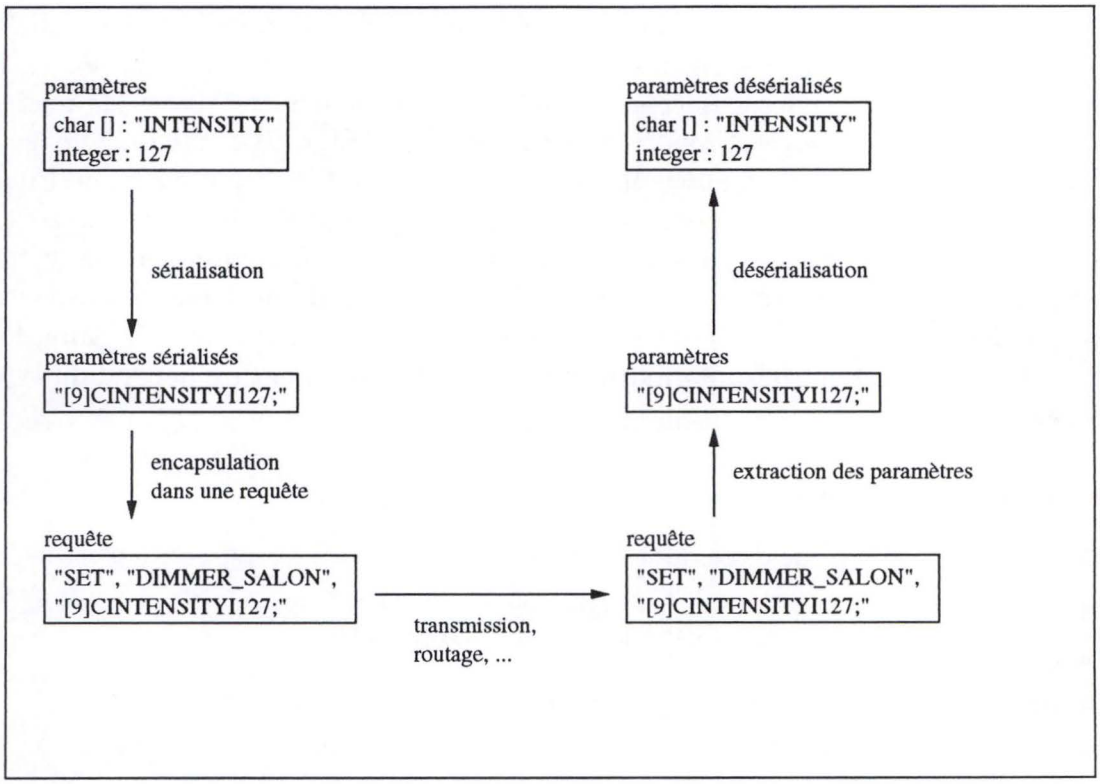

FIG. 4.4 – *Serialisation et désérialisation de paramètres.*

Structures des réponses

Une réponse est un message dont le type est REPONSE. Une réponse correspond toujours à une requête. Sa structure est la suivante:

- Code d'erreur;
- Résultat de l'opération;
- Paramètres de l'erreur;
- Identifiant de la requête associée.

Le champ *Code d'erreur* indique si l'opération s'est bien déroulée ou si une erreur s'est produite. Le cas échéant, le champ *Paramètres de l'erreur* peut contenir une description plus détaillée de l'erreur (cela dépendant du type d'erreur). Le champ *Résultat de l'opération* contient les informations obtenues suite à l'exécution de la requête. Par exemple, si la requête était l'interrogation de l'attribut d'un objet, la réponse contiendra la valeur de cet attribut s'il existe. Le résultat de l'opération est retourné sous forme sérialisée (cf. paragraphe précédent).

Structure des notifications

Une notification est un message dont le type est NOTIFICATION. Contrairement à une réponse, une notification ne correpond pas à une demande explicite (bien sûr, il a fallu s'enregistrer pour recevoir cette notification — cf. chapitre traitant du modèle objet), mais elle peut survenir n'importe quand. La structure d'un tel message est la suivante:

- Type de notification;
- Identifiant de l'objet émetteur de la notification.

Les notifications que l'on peut considérer au vu de ce qui a été présenté jusqu'ici sont 'ATTR_CHANGE' (indiquant le changement de la valeur d'un attribut d'une *ObjectInstance*) ainsi que 'DESTROYED' indiquant qu'une instance a été supprimée.

4.3 Architecture

L'architecture (illustrée à la figure 4.5) du framework repose sur une multitude de processus et de structures de données. Cette section va présenter ceux-ci et la façon dont ils interagissent pour arriver aux fonctionnalités décrites ci-dessus.

Les principaux processus qui entrent en jeu sont le serveur, le routeur, le moniteur et les processus de connexions. Les structures de données sont la table de routage, la table des abonnés, la file de messages (FIFO) et les tampons des processus de connexions. Comment tout cela s'agence-t-il? Prenons un serveur ou un client et voyons comment se passe sa connexion au framework ...

4.3.1 Serveur

Premièrement, le client (le raisonnement est similaire pour un serveur) se connecte au framework via un port (cf. [16]) écouté par le processus serveur. A ce moment, ce dernier est réveillé et accepte la connexion (si le nombre maximum de connexions configuré n'est pas atteint). Le serveur vérifie si la connexion est valide (vérification de l'adresse IP du client, par exemple). Si c'est le cas, un "gestionnaire de connexion est créé".

Un gestionnaire de connexion (détaillé à la figure 4.6) est un ensemble de deux processus chargés de lire et écrire sur la connexion.

- Le processus de lecture est en veille jusqu'à ce que des données soient arrivées sur la connexion. Il lit alors ces données et recompose des messages (dont la structure a été décrite ci-dessus). En effet, les messages peuvent avoir été segmenté par le protocole TCP. Chaque message reconstitué est placé dans la file de messages du routeur.

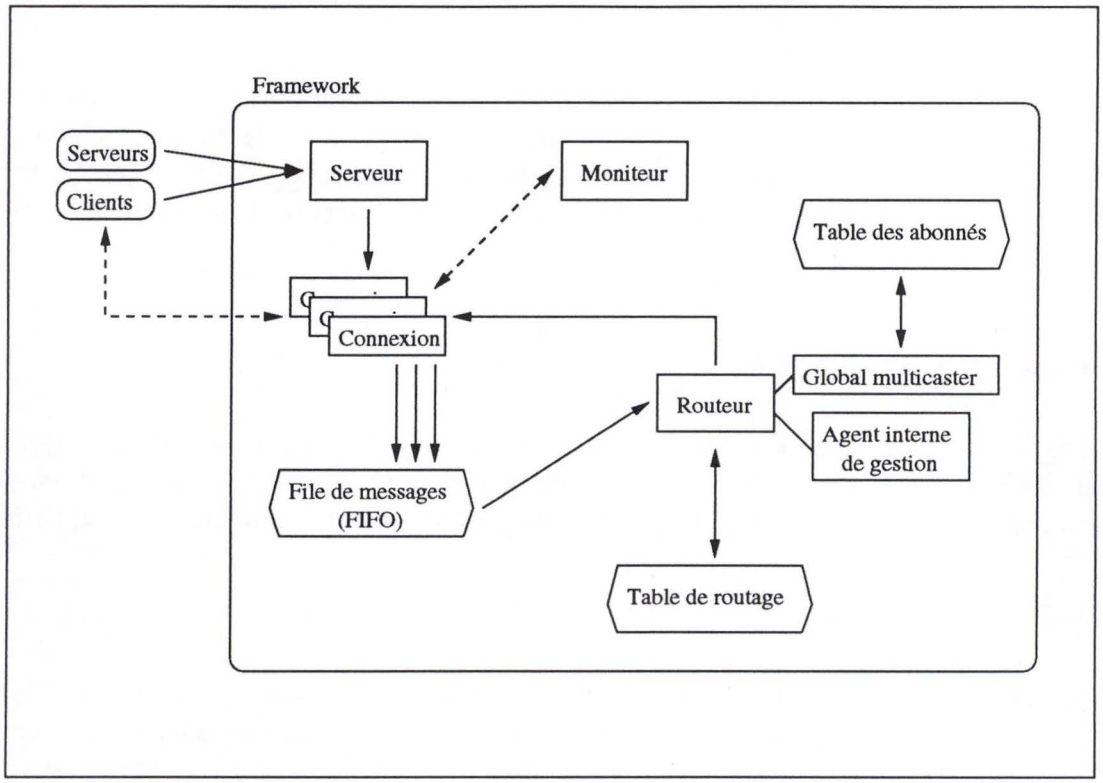

FIG. 4.5 – *Architecture du framework.*

- Le processus d'écriture reçoit des messages via une file FIFO. Il est en veille tant que la file est vide (ceci est réalisé à l'aide d'un sémaphore qui compte le nombre de messages de la file). Lorsqu'il y a au moins un message dans la file, le processus d'écriture le transmet sur la connexion. Nous verrons plus loin l'intérêt de cette file en entrée du processus décriture.

Il existe un gestionnaire par connexion. Ceci implique que la file de messages du routeur est accédée de façon concurrentielle par les n processus de lecture des connexions et par le processus du routeur. Il s'agit d'un problème de type producteurs-consommateur (figure 4.7) qui nécessite l'utilisation de moyens de synchronisation. En effet, il faut d'une part empêcher deux producteurs d'écrire un message simultanément, ce qui pourrait laisser la structure de messages dans un état indéterminé (dépendant de l'implémentation de cette structure): compteur de messages, pointeur de message courant, etc. D'autre part, il faut empêcher que le consommateur lise un message qui est en cours d'écriture par un producteur, ce qui pourrait entraîner la lecture d'un message incomplet.

Les primitives d'écriture et de lecture sont les suivantes:

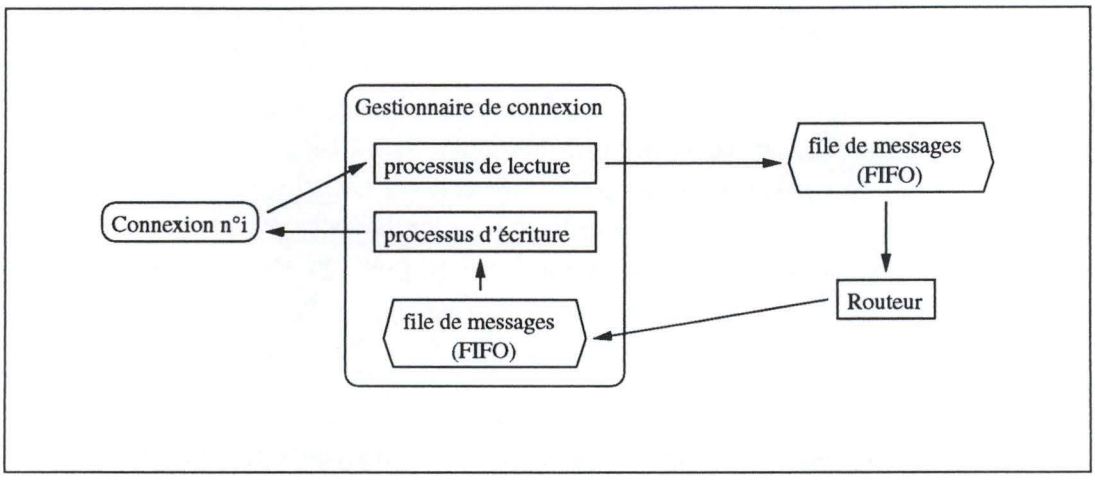

FIG. 4.6 – *Détail d'un gestionnaire de connexion.*

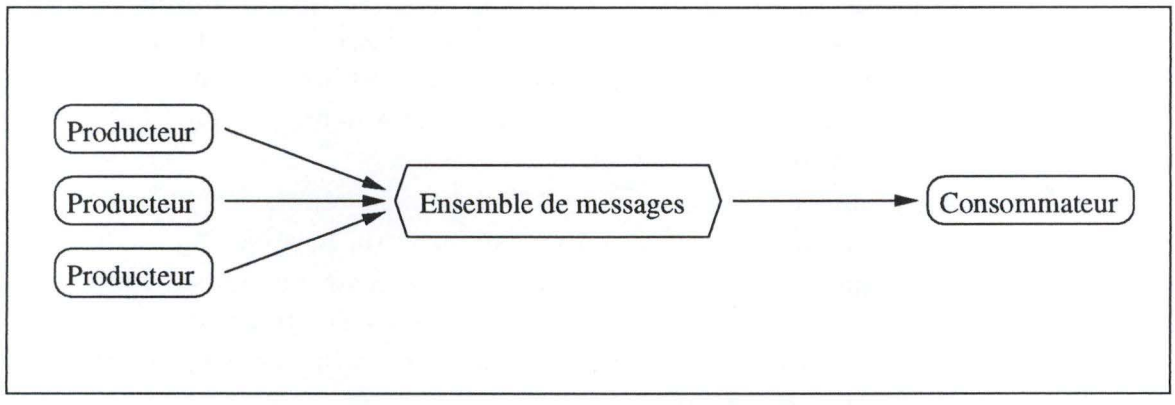

FIG. 4.7 – *Problème de type producteurs-consommateur.*

```
msg-write:
  mutex.wait();
  push(msg);
  semaphore.signal();
  mutex.signal();

msg-read:
  semaphore.wait();          (a)
  mutex.wait();              (b)
  pull(msg);
  mutex.signal();
```

Ainsi, l'accès concurrentiel entre les producteurs est empêché. De plus, le routeur (consommateur) ne peut lire quand un producteur écrit de sorte que la lecture d'un message incomplet est impossible. Remarquons que l'usage du sémaphore au sein des primitives d'écriture et de lecture sert uniquement à compter le nombre de messages de la file et à bloquer le consommateur au cas ou celle-ci est vide. Le test du sémaphore dans la primitive de lecture peut se faire en dehors de la section critique car on sait que le consommateur est unique, c.-à-d. le sémaphore ne peut qu'augmenter entre les lignes (a) et (b).

4.3.2 Routeur

Une fois qu'un message est arrivé entre les mains du routeur, celui-ci adopte différentes techniques de routage.

1. Le message est une **requête**: il s'agit de trouver la connexion qui a enregistré ("enrol") l'objet mentionné dans la requête. Ceci se fait en consultant la table de routage. Une fois la connexion identifiée, le routeur écrit la requête dans la file de messages du processus d'écriture de cette connexion. L'utilisation de cette file de messages est nécessaire pour éviter que le routeur soit bloqué dans l'attente que le message soit effectivement écrit sur la connexion.
2. Le message est une **réponse**: il s'agit d'émettre la réponse vers la connexion identifiée par le champ "*identifiant du destinataire*" du message.
3. Le message est une **notification**: Le routeur s'adresse au "global multicaster". Celui-ci émettra n requêtes vers les abonnés à l'objet émetteur. Remarquons que le multicaster fait partie du même processus que le routeur (il n'y a pas de problème de temps d'attente dans ce cas).

4.3.3 Moniteur

Le processus Moniteur vérifie régulièrement les gestionnaires de connexions et teste si ceux-ci sont toujours actifs. Cette opération est nécessaire pour permettre la libération des ressources nécessaires à la création de nouvelles ressources.

Si un client ou un serveur a fermé sa connexion avec le framework, il faut supprimer le gestionnaire qui était chargé de celle-ci. En effet, ce gestionnaire n'est plus utile d'une part et le nombre de gestionnaires est limité (par configuration) d'autre part.

Le processus Moniteur est également utilisé pour interroger régulièrement les serveurs connectés et vérifier s'ils sont toujours actifs (dialogue "ARE YOU UP?" — "YES"). Si ce n'est pas le cas, le serveur concerné devra être tué et re-démarré (cf. *Contrôle des serveurs*, ci-dessus).

4.4 Serveurs importants

Cette section contient la description de quelques serveurs dont le rôle est particulièrement important.

4.4.1 Le serveur *LonTalk*

Le serveur LonTalk est le serveur responsable de la gestion des objets représentant les noeuds et les connexions, physiques ou virtuels, d'un réseau LonWorks. C'est donc au sein de ce serveur que sont implémentées les requêtes cachées par les objets et qui agiront effectivement sur les objets du réseau.

En fait, le serveur LonTalk contient un *ObjectManager* (décrit au chapitre 3) qui reçoit les requêtes routées par le framework. Les *ObjectClasses* qu'il détient sont décrites au sein du code (elles ne proviennent pas de fichiers de description externes) de même que les procédures qui servent d'accesseurs ou qui sont cachées par les méthodes (celles-ci font également partie intégrante du serveur). Il existe deux types d'*ObjectClasses*: l'une représente un noeud basique et l'autre représente une connexion de variables. La représentation d'un noeud à la forme suivante:

```
objectclass NODE {
  integer NeuronID;
  void Reset();
  (...)
}
```

Comme on l'a décrit au chapitre 3, la variable stockée *NeuronID* matérialise le lien entre la représentation du noeud, à savoir, l'*ObjectInstance*, et le noeud physique. La méthode *Reset* permet de ré-initialiser le noeud et cache une procédure appelée *Reset* implémentée dans le code même du serveur LonTalk.

La représentation d'une connexion de variables à la forme de l'*ObjectClass* décrite ci-dessous:

```
objectclass CONNECTION {
  boolean AddVariable(char [], char []);
  boolean RemoveVariable(char [], char []);
  char [] [] ListOfVariables();
  (...)
}
```

La méthode *AddVariable* permet l'ajout d'une variable à la connexion. Il faut lui passer le nom de l'objet auquel appartient cette variable et le nom de la variable elle-même. L'ajout d'une variable correspond, comme on a pu le voir au chapitre 3, à l'enregistrement de l'instance de 'CONNECTION' comme *listener* de l'objet qui détient la variable connectée. La méthode *RemoveVariable* permet de supprimer une variable de la connexion. Enfin, la méthode *ListOfVariables* permet de renvoyer la liste des variables constituant la connexion.

Il faudrait distinguer les connexions de variables qui s'étendent sur plusieurs serveurs, i.e. dans lesquelles intervient au moins un objet d'un serveur différent de celui qui gère la connexion, des connexions de variables locales. En effet, ces dernières peuvent reposer sur le mécanisme de propagation de variables-réseau défini dans le protocole LonTalk.

4.4.2 Le serveur de gestion du framework

Le serveur de gestion du framework est un serveur interne (il apparait comme les autres serveurs aux "yeux" des clients, mais il fait partie intégrante du framework) qui est destiné à lancer un processus serveur, à le stopper, à effectuer des requêtes sur les tables du framework, etc.

Ce serveur sera identifié par le nom "." (point). Et disposera d'un certain nombre d'objets. Quels sont les objets publiés par le serveur de gestion du framework?

Objet framework

L'objet framework est l'objet référant tous les autres objets du serveur de gestion. Son ObjectClass est:

```
objectclass FRAMEWORK {
  void Start(char []);
  void Stop();
  integer NumServeur();
  char [] Serveur(integer);
}
```

La méthode *Start* permet le lancement d'un serveur (dont le framework doit avoir connaissance, c.-à-d. décrit et identifié dans un fichier de configuration). La méthode *Stop* déclenche l'arrêt du framework. Les méthodes *NumServeur* et *Serveur* permettent d'accéder à chacun des objets mis à disposition par le serveur interne du framework. Ceux-ci sont décrits ci-dessous.

Serveurs connectés

Chaque serveur connecté au framework est représenté par une instance de l'Object-Class suivante:

```
objectclass SERVEUR {
  char [] NAME;
  void Stop();
  integer NumObject();
  char [] Object(integer);
}
```

Le champ *NAME* est le nom fourni au framework par le serveur à l'aide l'opération **Bind**. La méthode *Stop* permet l'arrêt du serveur. Les méthodes *NumObject* et *Object* permettent de connaître les objets enregistrés par ces serveurs (à l'aide de l'opération **Enrol**).

Le serveur de gestion du framework permet donc à un client de reconstituer l'arborescence illustrée à la figure 4.8

4.4.3 Le serveur *Remote*

Dans le cas où le système est composé de plusieurs frameworks, on peut être amené à gérer des objets d'un framework (c.-à-d. publié sur un framework) à partir d'un autre. Dans ce cas, un client connecté à un framework pourrait manipuler un objet situé sur l'autre framework de manière parfaitement transparente.

Le mécanisme utilisé pour réaliser ce comportement repose toujours sur l'opération "**Enrol**" effectuée par un serveur. Voyons un exemple pour développer le mécanisme: Supposons que nous ayons deux frameworks, nommés FMK_A et FMK_B. Sur FMK_A, on voudrait rendre accessible un objet OBJ_1 situé sur FMK_B (c.-à-d. géré par un serveur connecté au framework FMK_B). A cet effet, on utilise un serveur particulier permettant un "**Enrol**" distant, SRV_REMOTE. Ainsi, quand un client s'adressera au framework FMK_A pour effectuer une requête concernant l'objet OBJ_1, cette requête sera dirigée vers le serveur SRV_REMOTE qui sera chargé de la communication avec le framework FMK_B par l'intermédiaire d'un client CLI_REMOTE connecté à celui-ci. CLI_REMOTE recevra donc des requêtes de SRV_REMOTE et les adressera à FMK_B. A ce moment, tout se passe comme si la requête avait été émise directement sur FMK_B.

Remarquons que des précautions sont à prendre lors de la conception du serveur *Remote*. On peut se trouver dans une situation où les deux frameworks considérés sont reliés par une ligne relativement lente et sur laquelle une connexion coûte cher (cas

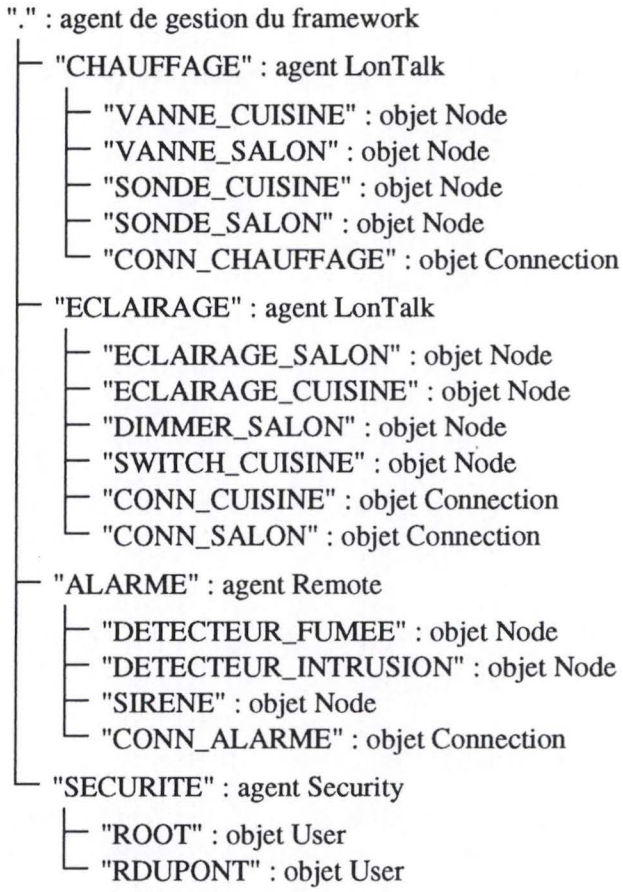

FIG. 4.8 – *Arborescence des serveurs et objets enregistrés.*

d'une ligne téléphonique). Dans ce cas, il faut éviter de rester connecté en permanence et privilégier les connexions lors de la réception de messages. On peut également envisager un système de tampon gardant les messages afin de les envoyer d'un bloc (c.-à-d. au sein d'une même connexion).

Cette publication d'objets distants peut poser des problèmes de cohérence au sein de la table de routage. En effet, il se peut qu'un objet publié sur le framework distant soit supprimé. Le client de ce framework va donc devoir rester à l'écoute de l'objet (afin de recevoir la notification de type 'DESTROYED') et indiquer au serveur *Remote* du framework local qu'un objet a été supprimé. Durant un certain temps (au moins égal au temps de connexion, de transmission de la notification et de mise à jour de la table de routage locale), il y aura incohérence entre les deux frameworks car un objet qui n'existe plus sur le framework distant sera publié sur le framework local. Ceci peut être évité en forçant le framework distant à attendre que le serveur *Remote* du framework

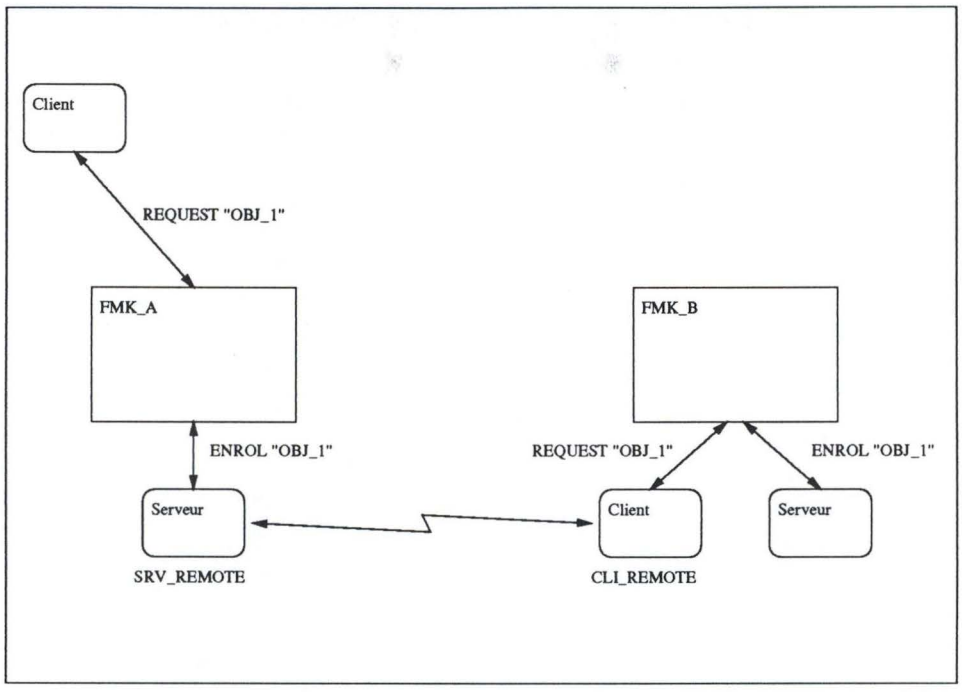

FIG. 4.9 – *Système comportant plusieurs frameworks*

local supprime l'objet publié avant de répondre au serveur (distant) que son objet a bien été supprimé (il s'agit en effet d'une opération avec acquittement). Cependant, le client associé au serveur *Remote* n'est pas sensé répondre à la notification 'DESTROYED' (puisque ce service ne nécessite pas de réponse). Le framework distant n'a donc pas de moyen de savoir quels sont les objets publiés à distance (et donc quels sont les clients qui les publient) ...

Un autre problème éventuel est la publication redondante d'objets. Un objet peut-il être publié deux fois sur le même framework via deux serveurs *Remote*? La réponse est non car cette publication repose sur l'opération "**Enrol**" et qu'un objet doit être unique dans la table de routage. Ainsi, si le premier (au sens chronologique) serveur *Remote* peut réussir à publier cet objet, le second verra sa requête refusée!

4.5 Conclusion

Le framework est une entité logicielle de gestion de messages relativement générique. Son extension à d'autres types de messages est aisée (au niveau du processus Routeur). De plus, des serveurs ayant des fonctions très diverses peuvent y être connectés sans que le framework ait à se préoccuper de la sémantique des messages qu'il véhicule.

Le framework constitue bien une solution aux besoins d'accès distant, d'organisation multi-clients et multi-serveurs et d'anonymat identifiés au chapitre 2. En outre, il est capable de détecter certains problèmes de fonctionnement des serveurs qui lui sont connectés (connexion rompue, file de messages pleine). Enfin, le système de serveur *Remote* permet son association avec des frameworks distants afin de partager des objets publiés localement.

Chapitre 5

Les interfaces (API)

5.1 Introduction

Maintenant que l'on a décrit l'organisation du système LonOS, c.-à-d. le framework et le modèle d'abstraction objet, il nous faut définir les interfaces (API[1]) mises à la disposition des développeurs de clients, de serveurs et d'objets.

Ce chapitre commence par exposer les outils à utiliser pour créer les procédures cachées derrière les accesseurs et les méthodes des *ObjectClasses* (voir chapitre 3). Ensuite, les interfaces de communication avec le framework, de manipulation distante des objets et de spécialisation seront présentées.

Quelques exemples de code source seront présentés au sein de ce chapitre car il s'agit bien d'interfaces de développement. Le langage utilisé est le *C++* car il est orienté-objet et assez répandu (on trouve des compilateurs sur la plupart des plate-formes). Cependant, l'utilisation d'un autre langage ne devrait pas poser énormément de problèmes (il faudra probablement créer une couche d'interfaçage, mais ceci n'est pas notre intérêt ici).

5.2 L'interface d'implémentation d'objets

On a vu au chapitre 3 que l'on pouvait représenter des ressources par des objets dont le type est défini via une *ObjectClass*. Cette *ObjectClass* décrit les attributs (stockés ou accédés), leur type (entier, booléen, complexe, etc.) ainsi que des méthodes. Le chapitre 3 expliquait également que ces méthodes ainsi que les accesseurs utilisés pour manipuler les attributs accédés étaient implémentés par des procédures ayant une interface particulière. Ces procédures pouvant se trouver au sein du code même d'un *ObjectManager* ou dans des librairies externes.

1. API: Application Programming Interface.

Cette section décrit l'interface qui permet de définir des accesseurs et des méthodes. On commence par les accesseurs:

5.2.1 Les accesseurs

Il existe deux types d'accesseurs. L'un pour lire la valeur d'un attribut (nous l'avions nommé *get*) et l'autre pour écrire cette valeur (appelé *set*). Leur forme était la suivante:

$$Set : ObjectInstance \times String \times Value \Rightarrow \_$$

$$Get : ObjectInstance \times String \Rightarrow Value$$

Comme la valeur passée à l'accesseur *set* ou renvoyée par l'accesseur *get* a un type bien déterminé, on passera à ces accesseurs un couple (<Type>, <Valeur>). Ainsi, on pourra utiliser un seul accesseur pour des types différents. Ce couple sera, dans les faits, matérialisé par un objet (toujours au sens *C++*) dénommé `CValue`. Celui-ci peut contenir une valeur dont le type n'est pas connu à priori. On peut l'interroger pour connaître le type de la valeur et pour adopter une conduite qui lui convient.

Le paramètre appelé *ObjectInstance* représente en *C++* l'*ObjectInstance* dont on veut lire ou écrire l'attribut. Il s'agit donc d'un objet lié à une *ObjectInstance* et qui permet de modifier les attributs de celle-ci ainsi que d'appeler ces méthodes. A cet effet, trois opérations sont mises à la disposition du développeur: `int SetAttr(char *, CValue *)`, `CValue * GetAttr(char *)` et `CValue * Call(char *, CValueList)`. Ce méthodes permettent respectivement de changer la valeur d'un attribut de l'*ObjectInstance*, récupérer la valeur d'un attribut et appeler une méthode.

On peut envisager d'autres méthodes, permettant de récupérer la définition de l'*ObjectInstance*, c.-à-d. son *ObjectClass*, et ainsi de connaître les attributs et les méthodes de celle-ci. Cependant, comme l'accesseur est développé pour une *ObjectClass* particulière, cette information est connue.

Il reste alors à implémenter l'accesseur. Par exemple, l'accesseur *SNVT_set* aura la forme:

```
int SNVT_set(CObjectInstance * pClInstance,
             char * pcAttrName,
             CValue * pClValue)
{
  int iNeuronID;

  // On récupère d'abord l'identifiant du noeud physique.
  iNeuronID= pClObjectInstance->GetAttr('NeuronID')->AsInteger();
```

```
  // On distingue ensuite le type de valeur.
  switch (pClValue->GetType()) {
  case CValue::INTEGER:
    // Il s'agit d'un entier.
    SNVT_set_integer(iNeuronID, pcAttrName, pClValue->AsInteger());
    break;
  case CValue::CHAR:
    (...)
    break;
  case CValue::ARRAY:
    (...)
    break;
  (...)
  }

  return 0;

}
```

Un accesseur de type *get* doit en plus renvoyer une valeur. Voici un exemple d'accesseur *SNVT_get*:

```
CValue * SNVT_get(CObjectInstance * pClInstance,
                  char * pcAttrName,
                  CValue * pClValue)
{
  CValue * pClReturnValue;

  (...)

  // Ici, pClValue est utilisé pour déterminer le type de retour.
  switch (pClValue->GetType()) {
  case CValue::INTEGER:
    iValue= SNVT_get_integer(iNeuronID, pcAttrName);
    pClReturnValue= new CValue(CValue::INTEGER, iValue);
    break;
  case CValue::CHAR:
    (...)
    break;
  case CValue::ARRAY:
```

```
        (...)
        break;
    (...)
    }

    return pClReturnValue;

}
```

5.2.2 Les méthodes

Pour les méthodes, le problème est similaire à celui des accesseurs. Il est toujours nécessaire de pouvoir manipuler l'*ObjectInstance* afin d'en lire ou écrire certains attributs, voir même d'appeler d'autres méthodes de celle-ci. Il faut en outre, gérer une liste de paramètres dont les types ne sont pas connus à priori (du moins du côté de l'*ObjectManager*). La forme d'une méthode est la suivante:

$$Method : ObjectInstance \times Value^n \Rightarrow Value$$

La liste des paramètres et leurs valeurs ($Value^n$) sera transmise par le biais d'un objet ($C++$) appelé `CValueList`. Cet objet permet de connaître le nombre de paramètres et de récupérer chacun d'eux. Chaque paramètre est une structure de type `CValue` déjà présentée à la section précédente. Bien entendu, le développeur connaît les paramètres et leurs types, donc, au sein de la procédure, il pourra extraire les divers paramètres et éventuellement tester si les types sont corrects. Comme dans le cas des accesseurs, on peut développer une procédure utilisée avec des signatures différentes.

5.3 Les interfaces de communication, de gestion d'objets et de spécialisation

Ces interfaces sont utilisées par les développeurs de clients et de serveurs afin que ceux-ci puissent se connecter au framework (interface de communication), manipuler des objets distants (publiés sur le framework et gérés par d'autres serveurs), voire effectuer d'autres opérations spécialisées.

Ces interfaces de programmation sont organisées en couches comme la section suivante le décrira. Ainsi, on devrait arriver à une structure équivalente à celle qui est illustrée à la figure 5.1.

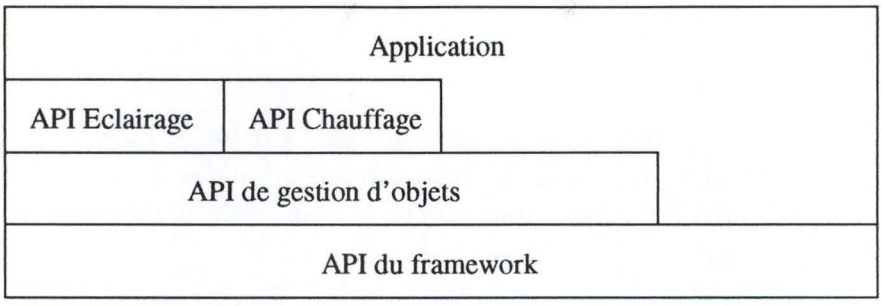

FIG. 5.1 – *Exemple d'empilement des couches.*

5.3.1 Le modèle

Le modèle d'organisation est une fois de plus un modèle en couches. Une fonction d'un certain niveau est réalisée à l'aide d'une ou plusieurs fonctions du niveau inférieur (ou d'un niveau inférieur) comme l'illustre la figure 5.2. Ainsi, si un client veut effectuer une requête sur un objet particulier dont il connait, à priori, le type, il va utiliser une fonction du niveau inférieur correspondant à la gestion de ce type d'objet. Les fonctions des niveaux supérieurs représentent donc des fonctions plus spécialisées et les fonctions des niveaux inférieurs des fonctions plus génériques.

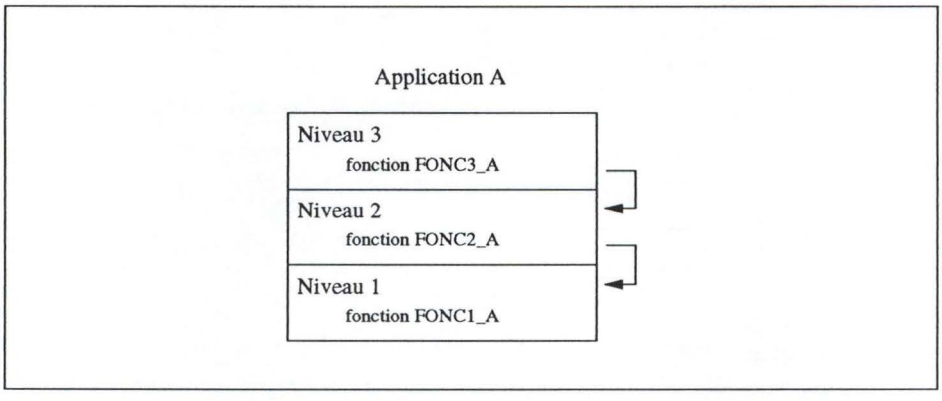

FIG. 5.2 – *Organisation en couches*

Par exemple, imaginons une application qui gère l'éclairage d'un immeuble. A un moment donné, l'application doit activer l'éclairage d'un pièce donnée pour une raison ou une autre (car l'utilisateur l'a demandé, par exemple). Elle connait l'identifiant de l'éclairage concerné (*ECLAIRAGE_BUREAU*). Au sein de l'application, il faudra effectuer un appel de fonction tel que (en prenant la même représentation qu'au chapitre

3):

```
ECLAIRAGE_BUREAU.SwitchON();
```

Si on se place dans l'architecture de gestion basée sur un framework et de multiples serveurs, cet appel de fonction cache la transmission d'une requête vers le framework. Ainsi, la fontion *SwitchON* utilise une fonction d'un niveau inférieur qui permet la transmission d'une requête vers le framework.

```
SendRequest('CALL', 'ECLAIRAGE_BUREAU', 'SwitchON');
```

Cette fonction (*SendRequest*) demande que la méthode *SwitchON* de l'objet *ECLAIRAGE_BUREAU*) soit appelée. Ce qui a pour effet l'activation de l'éclairage physique.

On constate effectivement que les fonctions du niveau *Gestion_Eclairage* sont plus spécialisées que les fonctions du niveau *Gestion_Requêtes*. Ces dernières peuvent être utilisées pour la transmission de toutes les requêtes à destination du framework. La figure 5.3 illustre cet exemple.

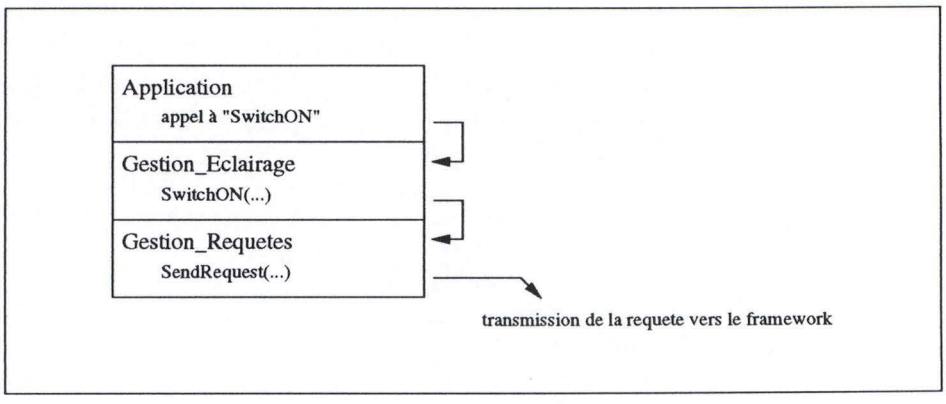

FIG. 5.3 – *Exemple d'application*

On appelera chaque niveau une **API**. Le niveau *Gestion_Requêtes* est l'interface de programmation du framework tandis que le niveau *Gestion_Eclairage* est l'interface de programmation des serveurs de gestion d'appareils.

5.3.2 L'API du framework

Fonctionnalités

L'API du framework doit permettre la transmission de requêtes et la réception des réponses. Elle doit en outre permettre le contrôle du framework, c.-à-d. connecter/déconnecter un serveur, tester si un serveur est présent, consulter les tables de routage, connecter/déconnecter un utilisateur (cf. gestion de la sécurité), etc.

On veut en outre que la gestion des requêtes soit asynchrone, c.-à-d. on ne veut pas que lorsqu'une application effectue une requête, elle soit bloquée jusqu'à la réception de la réponse à celle-ci !

Emission d'une requête et réception de la réponse

Une première solution consiste à utiliser une fonction "call-back"[2], c.-à-d. une fonction qui sera appelée lors de la réception de la réponse. Ainsi, l'application effectue la transmission de sa requête via l'API du framework et récupère immédiatement la main dès que la transmission est terminée. Une fois que la réponse arrive, l'API du framework effectue un appel à la fonction "call-back" spécifiée. Cette solution nécessite l'utilisation de plusieurs "threads" et permet un asynchronisme efficace mais oblige le développeur de l'application à s'occuper des problèmes de concurrence entre processus. En outre, cette solution est gourmande en ressources car elle nécessite la création de nombreux processus légers (*threads*) qui se terminent en un temps relativement court. Cette première solution est illustrée à la figure 5.4.

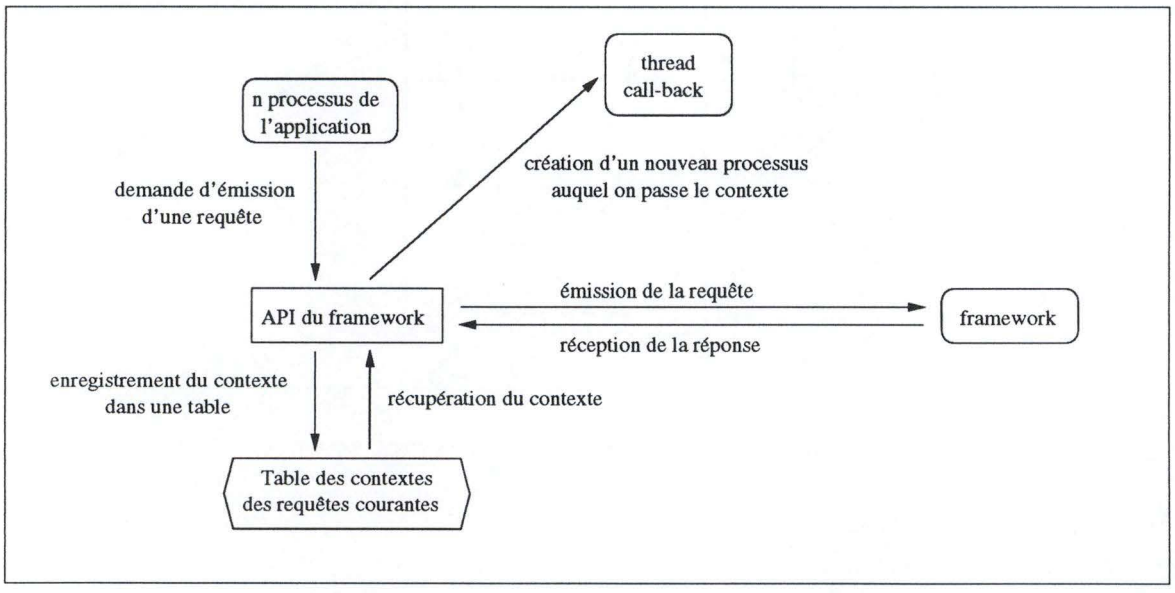

FIG. 5.4 – *Première solution asynchrone.*

Ce n'est pas cette solution qui a été retenue, mais la suivante (illustrée à la figure 5.5): Chaque fois qu'un message parvient à l'API, en provenance du framework, ce message est placé dans une file de messages. Cette file sera consultée par les processus

2. C'est la solution employée dans l'implémentation actuelle de SOW — SML Object Wrapper — de la suite logicielle OpenMaster.

de l'application. Les seuls problèmes de synchronisation sont ceux qui concernent l'accès concurrentiel à cette file. Ils seront gérés au sein de l'interface afin de décharger le développeur de cette préoccupation.

Remarquons également qu'il est nécessaire que l'émission de requêtes s'effectue de manière non bloquante, c.-à-d. lorsque l'application effectue un appel de la procédure d'émission d'une requête, celle-ci rende la main "immédiatement". Comme pour les processus d'écriture sur une connexion dans le framework, on utilise une file de messages. L'organisation de l'API du framework devient donc celle sui est illustrée à la figure 5.5.

La résolution des problèmes de synchronisation sur les files de messages impliquées dans le fonctionnement de l'API du framework est similaire à celle développée dans le chapitre 4 au niveau des processus de connexion et du routeur.

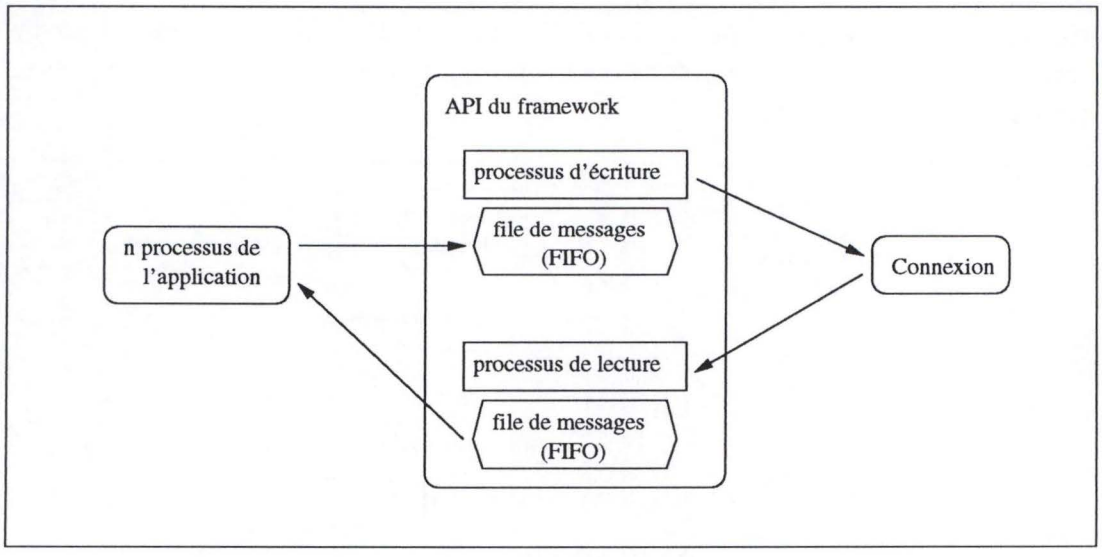

FIG. 5.5 – *Seconde solution asynchrone.*

Réception d'une notification

Le traitement d'une notification est basé sur le même principe que celui décrit ci-dessus. La différence la plus importante est que ce type de message n'est pas demandé explicitement. Ainsi, si une réponse peut être identifiée par le numéro de requête qui lui a donné lieu, ce n'est pas le cas d'une notification.

5.3.3 L'API de gestion d'objets

Fonctionnalités

L'API de gestion d'objets doit permettre de manipuler les objets suivant le modèle de gestion précédemment décrit. Il faudra donc pouvoir obtenir la description d'une instance d'objet, c.-à-d. son *ObjectClass*, effectuer des *Set* et des *Get* sur les attributs stockés ou accédés, appeler les méthodes de l'objet (locales ou distantes), etc.

Ainsi, l'API de gestion des objets est une sur-couche de l'API du framework qui permet un dialogue avec les agents capables de gérer des objets selon le modèle décrit précédemment.

Les requêtes que l'on peut émettre vers ces agents sont:

- CALL: appel d'une méthode;
- SET: modification de la valeur d'un attribut;
- GET: interrogation de la valeur d'un attribut;
- TYPE: description d'un objet (retourne son *ObjectClass*);
- CREATE: création d'un objet;
- DESTROY: suppression d'un objet.

Le but de l'API de gestion des objets est d'encapsuler ces requêtes dans des procédures facilement utilisables. Par exemple, en *C*, la requête "SET" peut être représentée par une procédure dont l'en-tête est

```
set(char * pcObjectName, char * pcAttrName, int iValue);
```

pour un attribut dont le type est *integer*.

Si l'interface est destinée à un langage de programmation orienté-objets, on peut fournir une classe qui regroupe toutes ces procédures et dont on crée une instance par objet (LonOS) qu'on utilise. Par exemple, en *C++*,

```
class CLonOS_Object
{
public:
  CLonOS_Object(char * pcObjectName);
  ~CLonOS_Object();
  void Set(char * pcAttrName, int iValue);
  void Set(char * pcAttrName, char * pcValue);
  (...)
  int IsInit();
  CAttrList * GetAttrList();
  CMethList * GetMethList();
```

```
private:
  char * pcObjectName;
  int iInit;
  CAttrList * pClAttrList;
  CMethList * pClMethList;
};
```

Cet objet serait chargé de l'interrogation de l'agent afin d'obtenir la structure de l'objet (*ObjectClass*). Il serait également à même d'effectuer toutes les requêtes dont on a parlé ci-dessus.

Le changement de l'attribut *Intensity* de l'object *ECLAIRAGE_BUREAU* dont on a parlé au chapitre 3 serait donc effectué par

```
ECLAIRAGE_BUREAU.Set('Intensity', 127);
```

où *ECLAIRAGE_BUREAU* est une instance de l'objet *CLonOS_Object* décrit ci-dessus.

5.3.4 Les API spécialisées

Les agents particuliers peuvent fournir des objets ayant de nombreuses fonctionnalités pour lesquels une interface spécialisée peut être fournie. Par exemple, l'interface représentée par l'objet *CLonOS_Object* de l'API de gestion des objets peut être enrichi par une classe qui est une interface pour un objet particulier (c.-à-d. pour une *ObjectClass* particluière).

Soit l'*ObjectClass* NODE_DIMMER décrite au chapitre 3. Le fabricant de matériel abstrait par cette *ObjectClass* peut également fournir une interface de programmation destinée au C++:

```
class CLonOS_Dimmer : public CLonOS_Object
{
public:
  void SetIntensity(int iValue);
  int GetIntensity();
};
```

Plus besoin de manipuler le nom de l'attribut *Intensity* et les méthodes *Set* et *Get*. Les méthodes *SetIntensity* et *GetIntensity* fournissent un moyen d'accès simplifié au maximum. Il est également possible de représenter des méthodes de l'*ObjectClass*, ce qui simplifie le passage de paramètres.

5.4 Conclusion

Ce chapitre à présenté les interfaces mises à la disposition des développeurs qui doivent étendre le système LonOS. Cette extension est possible au niveau de l'abstraction objet en permettant de définir des objets spécialisés (ceci vient donc compléter le chapitre 3) et des interfaces de manipulation propres à ceux-ci. L'extensibilité du système LonOS est également due au fait que l'on peut définir aisément un nouveau client. Des interfaces sont en effet fournies pour sa connexion au framework et pour la manipulation des objets publiés sur celui-ci.

On a également vu dans ce chapitre la méthode utilisée pour fournir l'asynchronisme dont il est fait mention dans le chapitre 2.

On pourrait étendre l'interface de gestion d'objets en y ajoutant un mécanisme de *proxy*. Celui-ci consisterait à conserver localement, c.-à-d. au sein du client, une instance de l'objet auquel on accède (ceci est communément appelé *shadow object*). Ainsi, lorsque le client souhaite accéder à l'objet, en lecture, c'est à l'objet local qu'il le fait (et ceci nécessite que cet objet local soit mis à jour lorsque l'instance "réelle", gérée par le serveur, est modifiée). En revanche, lorsqu'il doit modifier un attribut de l'objet ou en appeler une méthode, c'est à l'objet distant qu'il s'adresse.

Chapitre 6

Exemple

6.1 Introduction

Afin d'illustrer les concepts introduits, ce chapitre va présenter l'utilisation du LONOS au sein d'une application fictive. A travers les différentes étapes du développement, nous verrons les modifications à apporter aux démarches déjà existentes.

6.2 L'application

L'application fictive est le contrôle d'accès à l'Institut d'Informatique. Actuellement, tout le monde peut entrer dans le bâtiment de l'Institut sans être inquiété. Suite à divers incidents et à la surcharge des salles informatiques due à la présence d'étudiants étrangers à l'Institut, le Conseil d'Administration décide d'opter pour une solution de contrôle d'accès basé sur une identification par carte magnétique ou par carte à puce.

Suite à l'installation du système, il devrait être possible de savoir qui est à l'Institut à un moment donné. Ceci permet également de donner un accès à certains étudiants hors des heures ouvrables habituelles et ceci dans le cadre d'un travail bien particulier. Les personnes étrangères qui doivent se rendre à l'Institut se verront remettre un badge activé pour l'occasion durant un laps de temps restreint leur permettant d'entrer.

Le Conseil d'Administration a fait appel à une société de consultance pour l'analyse des besoins et la proposition d'une solution complète. Après moult études et présentations de transparents, un rapport est fourni décrivant le développement de l'application suivant les étapes suivantes:

1. Développement du matériel: celui-ci est demandé à une société spécialisée dans le contrôle d'accès pour les serrures électro-mécaniques tandis que pour les lecteurs de cartes, il existe déjà une solution qui convient (il suffira de les acheter).

2. Développement de l'application de contrôle propre à l'Institut d'Informatique (voir la figure 6.1). Cette application reposera sur le système LonOS, ainsi le développement sera restreint au contrôle d'accès au sein de l'Institut d'Informatique et à la gestion des autorisations individuelles.
3. Dans le futur, l'application pourra être étendue à la sécurité générale du bâtiment (détection d'incendie et d'intrusion) ainsi qu'à la gestion de l'éclairage.

La technologie utilisée sera la technologie LonWorks pour les raisons suivantes: elle facilite l'interopérabilité des noeuds (ceci permettra entre autres d'intégrer les lecteurs de cartes déjà développés); il est possible de transmettre des trames LonTalk sur un réseau Ethernet ce qui permet l'installation sans grandes modifications de l'infrastructure existante.

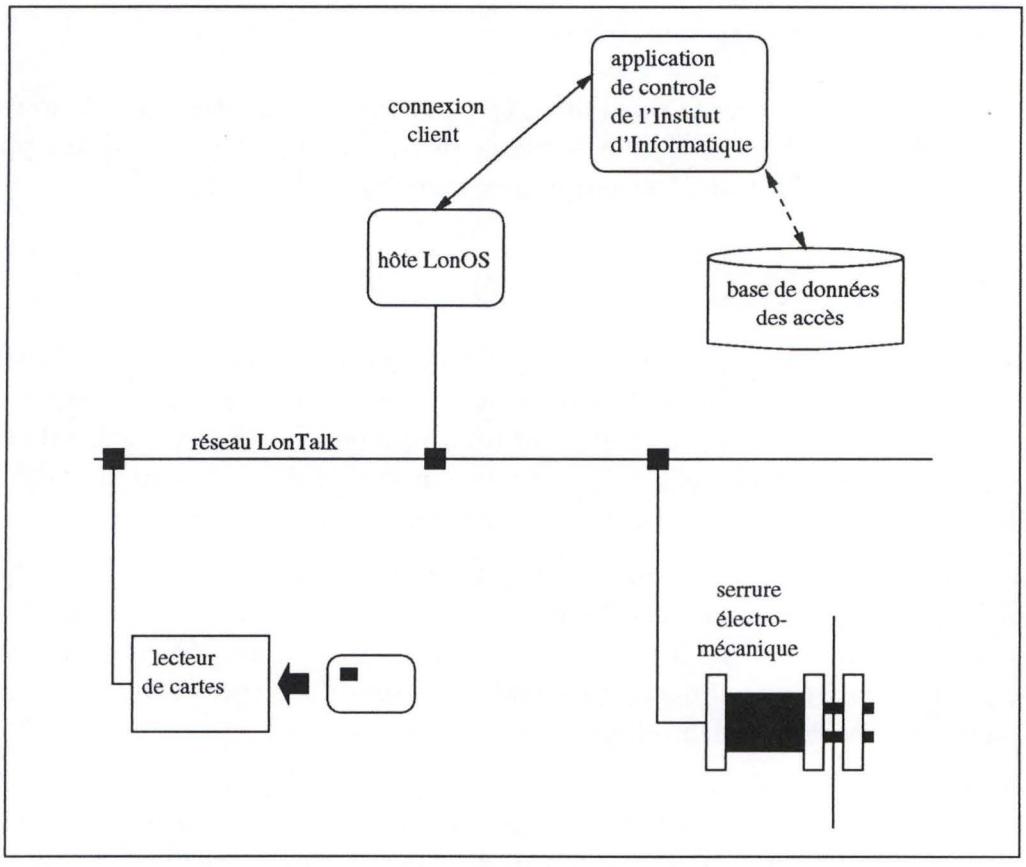

FIG. 6.1 – *Une partie de l'application.*

6.3 Développement du matériel

En conclusion, le développement de matériel destiné au système LonOS comprend trois phases:

1. Développement du matériel pur et de son *firmware*;
2. Développement d'*ObjectClass*;
3. Développement d'interfaces de programmation spécifiques.

Développement du matériel pur et de son firmware: Cette première phase consiste à développer le matériel électronique de l'appareil. Il s'agit ici d'une commande de serrure électro-magnétique. L'architecture de ce matériel correspond à celle donnée au chapitre 1 sous l'appelation "Schéma d'un noeud typique" (figure 1.3). Au coeur d'un noeud se trouve un processeur LonWorks (*NeuronChip*) qui interagit avec des périphériques d'une part et qui est en communication avec les autres noeuds par le réseau d'autre part. Les périphériques se limitent ici à une commande d'électro-aimant pour la serrure électro-mécanique.

La commande de ces périphériques est déclenchée par des messages de propagation de variables reçus par le réseau. En outre, des traitements internes peuvent être requis au sein d'un noeud. Ceci relève du *firmware* de gestion du noeud.

Développement d'ObjectClass: Le développement d'*ObjectClass* comprend une interface (un fichier texte dont la syntaxe a été décrite au chapitre 3) décrivant les attributs et les méthodes ainsi qu'une implémentation (ensemble de procédures regroupées dans une librairie partageé — voir les sections 3.5 au chapitre 3 et 5.2 au chapitre 5).

Cette *ObjectClass* sera utilisée par l'agent LonTalk chargé de gérer le noeud physique (la serrure électro-magnétique). L'interface de l'*ObjectClass* qui accompagne le matériel est:

```
objectclass NODE_LOCK extends NODE {
  void Unlock();
  void Lock();
}
```

Développement d'interfaces de programmation spécifiques: Enfin, la société a décidé de fournir une interface de programmation (API) spécifique destinée au pilotage de la serrure électro-mécanique. Cette interface prend la forme d'une classe *C++* étendant la classe *CLonOS_Object* fournie par l'interface de gestion des objets (voir chapitre 5). Cette classe a la forme suivante:

```
class CLonOS_Lock : public CLonOS_Object
```

```
{
public:
  void Unlock();
  void Lock();
};
```

Une fois que l'on a ces différentes entités logicielles, on dispose d'une configuration équivalente à celle qui est illustrée à la figure 6.2.

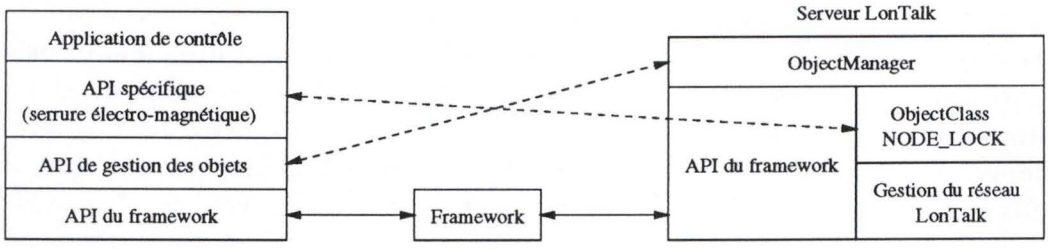

FIG. 6.2 – *Intégration des différents développements relatifs au matériel.*

Les deux dernières phases (développement de l'*ObjectClass* et développement d'une API) sont cependant facultatives. Le système LONOS est capable de gérer des objets pour lesquels aucune *ObjectClass* n'a été fournie. En effet, lors de la compilation du *firmware* d'un noeud (à l'aide d'un compilateur fourni par Echelon), une information[1] de description du noeud et de ses variables-réseau est générée. Cette information est utilisée comme substitut d'une *ObjectClass* basique ("sans valeur ajoutée").

Les lecteurs de cartes acquis sont fournis avec l'*ObjectClass* suivante dérivée du fichier *.XIF* généré à la compilation du *firmware*:

```
objectclass NODE_CARD_READER extends NODE {
  char [] Identifier;
}
```

6.4 Développement de l'application

Il s'agit maintenant de développer l'application qui effectuera réellement la gestion du système de contrôle d'accès. On veut que la porte principale de l'Institut d'Informatique ne s'ouvre qu'après présentation d'une carte à puce, validation de son contenu et uniquement si cette carte possède un accès durant la tranche horaire donnée. En outre,

1. Cette information est dénommée XIF — eXternal Interface File —

on veut que soit enregistré dans un fichier l'événement ouverture de la porte associé à l'identifiant de la carte (qui est aussi l'identifiant de la personne qui s'est présentée).

L'implémentation de cette application est relativement simple et repose sur l'utilisation des interfaces de programmation fournies avec les serrures électro-mécaniques et les lecteurs de cartes à puce. L'application, qui a connaissance des identifiants des différents appareils connectés au réseau, indique qu'elle est intéressée par la réception des notifications en provenance de l'objet `DOOR_CARD_READER` dont l'*ObjectClass* est `NODE_CARD_READER`. La notification qui intéresse l'application est celle qui concerne la modification de l'attribut `Identifier` (défini dans l'*ObjectClass* `NODE_CARD_READER`) qui contient l'identifiant de la carte. Chaque fois que cet attribut change, cela signifie soit la présentation d'une carte soit son retrait (la valeur de l'attribut `Identifier` est alors mise à zéro.

Lors de l'insertion de la carte, l'application effectue une recherche dans une base de données d'utilisateurs et vérifie l'authenticité de la carte (la validité de l'identifiant) ainsi que la plage horaire autorisée. Si tout ceci est valide, l'application s'adresse maintenant à l'objet `DOOR_LOCK` de type `NODE_LOCK` et appelle la méthode `Unlock`. Elle initialise en même temps un *timer* qui servira à déclencher la fermeture de la porte après un certain temps (par le biais de la méthode `Lock`) et ceci même si la carte reste insérée dans le lecteur (on peut d'ailleurs imaginer que cette carte soit "avalée" par le lecteur si elle y reste un temps trop long).

D'autres fonctionnalités ont été intégrées dans le cadre de cette application, mais ce bref aperçu donne déjà quelques idées de la façon dont le projet est développé.

6.5 Installation

L'installation du système de contrôle (c.-à-d. le système LonOS, l'application de contrôle et le matériel) comprend également deux phases: une installation matérielle (placement des noeuds et du réseau) et une installation logicielle (configuration du système informatique).

La configuration du système LonOS consiste à indiquer quels sont les agents à charger (ici, on considérera un seul agent LonTalk) et à créer les instances d'objets: `DOOR_CARD_READER`, `DOOR_LOCK`, etc. Cette création peut se faire sur base d'une découverte automatique du réseau implémentée au sein de l'agent LonTalk ou en indiquant l'adresse du noeud (c.-à-d. son *NeuronID*). Au terme de cette configuration, une série de tests peut déjà être menée[2]: on peut modifier les variables des noeuds et observer si

2. A l'aide d'un outil qui fera à terme partie du LonOS mais qui n'est pas encore développé à l'heure actuelle: un outil de monitoring permettant de visionner tout ou partie du réseau sous forme graphique et d'entreprendre des modifications sur les noeuds visibles, ... Cet outil devrait ressembler à l'application Monitor de la suite logicielle OpenMaster

le résultat est bien celui escompté (par exemple, ouverture ou fermeture de la porte).

Il reste alors à connecter l'application de contrôle munie d'une base de données de tests et à vérifier si l'introduction d'une carte valide donne bien lieu à l'ouverture de la porte.

6.6 Evolution

Le système ainsi mis en place est prêt à évoluer. Il "suffit" d'ajouter les noeuds nécessaires, par exemple, à la gestion de l'éclairage, et à développer l'application destinée à les contrôler. Pour des raisons de performances ou de charge du réseau LonTalk, il peut être nécessaire de diviser celui-ci et d'utiliser plusieurs interfaces avec la machine hôte.

Enfin, si d'autres facultés souhaitent également mettre en oeuvre un tel système de gestion d'accès, de contrôle d'éclairage, etc. Il leur est loisible d'utiliser le ou les hôte(s) LonOS afin de bénéficier des mêmes applications, par exemple.

Conclusion

Pour conclure, voyons quels sont les apports du système LonOS au regard des besoins exprimés dans les premiers chapitres de ce document. On a vu dans l'introduction qu'il était intéressant d'adjoindre aux réseaux industriels et domotiques des possibilités de connexions entre eux et avec Internet ou des réseaux locaux (LAN), ceci dans le but d'en permettre l'exploitation et l'administration distante ainsi que l'interopérabilité.

Ensuite, au chapitre 1, des exemples d'applications tels que le contrôle de discothèques, de centres de réprographies et plus génériquement de réseaux de points de contrôle aux fonctionnalités variées (exemple d'utilisation en domotique) ont été présentés. Ce chapitre dégageait ainsi une première difficulté relative aux réseaux de points de contrôle: la diversité aussi bien des protocoles de communication que des fonctionnalités offertes par chaque noeud particulier. La technologie LonWorks a été présentée comme un premier pas dans la résolution de ce problème, avec comme leitmotiv "l'interopérabilité des noeuds" par la définition d'un protocole de communication appelé LonTalk. En fin de chapitre, l'exemple de l'application OpenMaster de la société française Bull a introduit les concepts d'administration de systèmes d'ordinateurs selon OSI. L'introduction de ces concepts permettra le développement d'analogies et de comparaisons avec l'exploitation de réseaux de points de contrôle.

J'ai ensuite repris plus formellement, au chapitre 2, les besoins auxquels le système LonOS devrait répondre dans un premier temps et certains besoins à développer dans un futur proche. Ces besoins étaient les suivants:

- une gestion orientée-objet,
- une organisation multi-clients, multi-serveurs,
- un accès distant,
- la transparence dans la localisation des objets (ce que nous avons nommé *anonymat.*

Les chapitres suivants ont alors présenté petit à petit des éléments de solution qui en fin de compte constituent la version actuelle du système LonOS.

Le chapitre 3 a présenté la gestion orientée-objet à la base du système LonOS. Celle-ci se décline en une série de concepts tels que les *ObjectClasses* et les *ObjectInstances*

qui correspondent aux classes et instances de l'approche objet classique mais qui ont été modifiées afin de représenter les ressources que l'on retrouve sur les réseaux de point de contrôle (noeuds et connexions de variables). Ces modifications portent les noms d'*Attributs stockés*, d'*Attributs accédés* et de *Méthodes*. La présentation de cette gestion orientée-objet est restée basée sur une découpe entre l'interface d'une *ObjectClass*, c.-à-d. un fichier texte décrivant les attributs et les méthodes, et son implémentation sous la forme de librairies dynamiques de procédures écrites en *C++* (et en Java dans le futur). Cette partie implémentation repose sur des interfaces de programmation (API) décrites au chapitre 5. Le chapitre 3 répondait donc au premier besoin mis en exergue au chapitre 2, à savoir un besoin de gestion orientée-objet.

Ensuite, il restait trois besoins: une organisation multi-clients/multi-serveurs, une possibilité d'accès distant et l'exigence d'anonymat. Le chapitre 4 décrit une solution à ces trois besoins: le framework. Inspiré de la suite logicielle OPENMASTER de BULL, le framework est l'entité centrale du système LONOS. Le framework permet l'interconnexion des clients et des serveurs du système, ainsi que le routage des messages que ceux-ci s'échangent. L'accès distant est fourni par l'utilisation de connexions TCP/IP entre les clients, les serveurs et le framework. L'anonymat repose sur le routage des requêtes. On a vu au, chapitre 2, que les clients manipulent des objets gérés par des serveurs. Ces manipulations cachent un dialogue de requêtes-réponses entre les clients et les serveurs, ces requêtes concernant des objets bien identifiés. C'est sur base de ces objets que le framework va diriger une requête d'un client vers le serveur qui gère l'objet spécifié. Ceci résoud donc le problème de la localisation des objets puisqu'un client ne doit plus connaître que l'identifiant d'un objet et pas le serveur sur lequel ce dernier se trouve.

On a ainsi répondu aux besoins d'exploitation présentés au chapitre 2. Il reste cependant quelques reproches à adresser au système LONOS:

1. Le système LONOS est en développement. Il ne bénéficie donc pas encore des nombreux tests qu'un système de gestion d'objets répartis tel que CORBA a pu encourir. Son développement est cependant nécessaire car les systèmes existant actuellement (à un coût raisonnable) n'offrent pas les fonctionnalités requises par l'exploitation de réseaux de points de contrôle telle qu'elle est conçue dans ce document;
2. Le système LONOS est adhérent à la technologie LonWorks. En effet, dans ce document, on a parlé de cette technologie uniquement, et notamment de la représentation des connexions de variables-réseau. Cependant, l'intégration d'un serveur/gestionnaire d'un autre réseau ne devrait pas poser de problèmes (tous mes tests on été effectués avec un appareil connecté à une ligne RS-232);
3. Ensuite, il existe une certaine redondance d'information entre les serveurs. En effet, si on utilise plusieurs serveurs de même type, ceux-ci vont chacun avoir des

définitions d'*ObjectClasses* locales (ceci n'est vrai que s'ils sont situés sur des hôtes différents). On pourrait envisager de placer une partie de cette information sur le framework;

4. Le problème de la chronologie des événements n'a pas été traité: cas où deux requêtes sont émises "simultanément" par un client; cas où une requête émise par un client arrive après une requête émise par un autre client à un temps ultérieur, etc.

Le système ainsi analysé possède déjà une implémentation partielle fonctionnant sous Linux. Seule la partie concernant l'écoute d'objets n'est pas implémentée. Mais le projet ne va pas en rester là et les fonctionnalités qui n'ont pas été détaillées, mais juste évoquées, auront l'occasion de voir le jour dans les mois qui viennent. En particulier, l'application de monitoring et d'administration, la gestion de transactions et la sécurité sont importants.

Annexe A

Pré-requis

La plupart des informaticiens connaissent ces concepts de nom. Cependant, il est bon de rappeler ici ce que ces notions recouvrent exactement. On pourra également se référer aux références suivantes: [19], [22] ainsi que dans [16] pour une utilisation sous les systèmes d'exploitation de la famille UNIX.

A.1 Threads

Le terme *thread* (*fil*) représente un chemin d'exécution dans un programme. Ainsi, n'importe quel programme a au moins un *thread* d'exécution. L'importance de la notion de *thread* est survenue avec les programmes *multi-threads*, c.-à-d. des programmes qui possèdent plusieurs chemins d'exécution. L'objectif d'un tel changement est la possibilité d'effectuer plusieurs traitements en parallèle en les distribuant sur différents *threads*. Ces *threads* pouvant eux-même être exécutés par des processeurs différents.

Un programme possédant plusieurs *threads* possède plusieurs contextes d'exécution. Ainsi, dans un programme n'ayant qu'un chemin d'exécution, ce contexte sera constitué des éléments suivants:

- Segment de code (programme);
- Pointeur d'instruction;
- Pile (stack);
- Segment de données;
- Autres informations (contenu de registres, etc.)

Un programme *multi-threads* possède une pile et un pointeur d'instruction par *thread*. En revanche, les segments de code et de données sont partagés. Ce qui peut entraîner des problèmes de concurrence à gérer avec les moyens de synchronisation qui suivent ...

A.2 Sémaphores

Un sémaphore est une variable sur laquelle deux opérations atomiques sont possibles: *wait* (ou *probeer*) et *signal* (ou *verhoog*). Ces deux opérations peuvent être décrites comme suit:

```
wait():
  while (semaphore <= 0);
  semaphore--;
```

et

```
signal():
  semaphore++;
```

Un sémaphore est partagé par plusieurs *threads* ou processus et utilisé à des fins de synchronisation. Par exemple, pour protéger une structure de données accédée de façon concurrentielle par ceux-ci. Un exemple de problème de concurrence typique est celui des producteurs-consommateurs dont une variante est traitée dans ce document.

A.3 Mutex

Un mutex ("mutual exclusion"), parfois abusivement appelé sémaphore binaire, est un sémaphore particulier dont la valeur initiale est 1.

Annexe B

Implémentation

B.1 Introduction

Cette annexe décrit quelques-uns des problèmes relatifs à l'implémentation du système LONOS sous Linux (implémentation actuelle). La structure de l'implémentation est brièvement décrite (diagramme des classes), puis quelques explications concernant la portabilité seront données. Enfin, la méthode utilisée pour charger dynamiquement des librairies sous Linux sera exposée et on dira un mot des outils de synchronisation employés.

B.2 Diagramme des classes

Le diagramme de la figure B.1 représente la hiérarchie des classes (*C++*) que l'on retrouve dans l'implémentation. La classe de base, appelée `CObject` dispose d'un compteur partagé par toutes les instances et destiné à contrôler la création et la destruction de celles-ci. La classe `CObject` permet en outre de gérer des messages à enregistrer (*log*).

Parmi les autres classes, on distinguera les moyens de synchronisation tels que les sémaphores (classe `CSemaphore`), les mutex (class `CMutex`) suivis des threads (classe `CThread`) et des librairies partagées (classe `CLibrary`). Ces classes font l'objet d'une section spéciale (la suivante).

La hiérarchie de classe `CMessage` définit les diverses structures de messages manipulées par le framework (`CRequest`, `CResponse` et `CNotification`). Si on souhaite en ajouter de nouvelles, il suffira d'étendre la classe `CMessage`.

Les classes `CConnection`, `CServer`, `CFmkServer` et `CFmkAPI` représentent des connexions (éventuellement spécialisées). `CFmkServer` est une classe `CServer` qui contient les processus *Routeur* et *Moniteur* décrits au sein du chapitre 4.

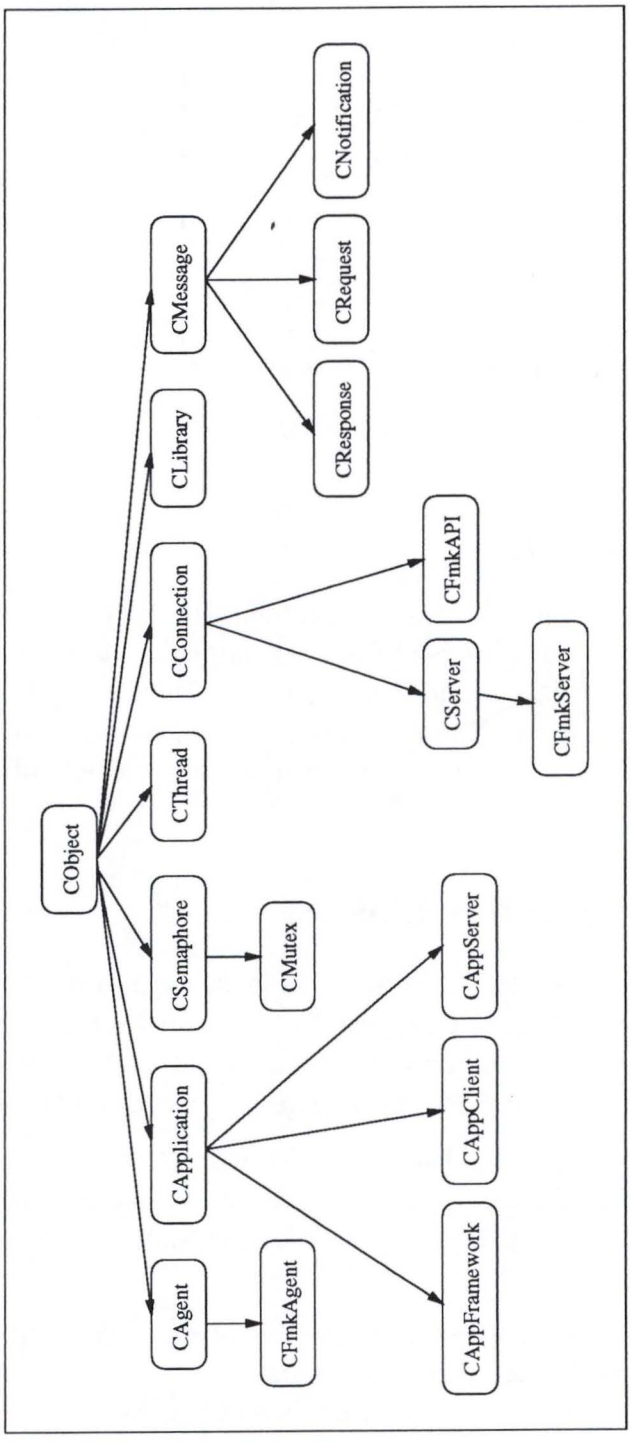

FIG. B.1 – *Diagramme des classes.*

Enfin, la classe `CApplication` effectue quelques initialisations comme la mise à zéro du compteur d'objets (voir la classe `CObject`) et la définition du flux de log.

B.3 Souci de portabilité

Lors de la conclusion du chapitre 2, on a fait mention d'un besoin de portabilité (souci commun avec de nombreux développements). Cette section présente la façon dont cette portabilité est améliorée au sein de l'implémentation du système.

Cette portabilité améliorée est assurée par l'utilisation de classes spécialisées pour les objets dépendant du système d'exploitation tels que les outils de synchronisation, les threads, les librairies partagées et les communications. Ces classes possèdent des interfaces immuables selon leur emploi (sémaphores, mutex, connexion, ...) mais ont une implémentation dépendant du système d'exploitation. Ainsi, pour porter le système d'une plate-forme à une autre, il ne devrait plus y avoir qu'à redéfinir l'implémentation de ces classes (la figure B.2 illustre ce souci).

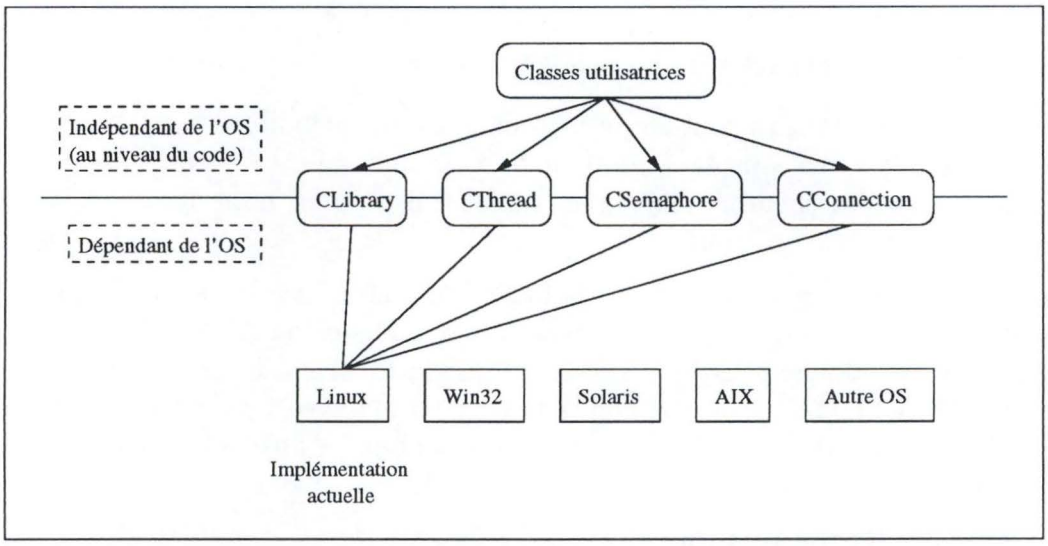

FIG. B.2 – *Souci de portabilité.*

B.4 Librairies partagées, threads et sémaphores

Cette section décrit l'utilisation des outils de synchronisation, des threads et des librairies partagées sous Linux. Ces dernières sont présentées de manière un peu plus détaillée car j'ai trouvé peu de documentation à leur sujet (mises à part les "man pages").

B.4.1 Librairies dynamiques

Il est possible, sous Linux, d'utiliser des librariries dynamiques en tant que "plug-ins", c.-à-d. c'est le programme lui-même qui indique au système quelles librariries il veut charger en mémoire et quels symboles il souhaite utiliser. C'est le cas, par exemple, si on dispose d'une librariries `libLonTalk.so` contenant la fonction `SNVT_set` que l'on veut appeler avec les paramètres `'node_dimmer'`, `'intensity'` et 127. La fonction `SNVT_set` a la signature suivante:

```
void SNVT_set(char * pcNodeID, char * pcVarID, int iValue)
```

Il s'agit précisément d'une des tâches de l'entité *ObjectManager* présentée au chapitre 3. L'*ObjectManager* est en effet responsable des fonctions cachées derrières les méthodes des *ObjectClasses*.

Ceci est rendu possible par une série de fonctions standard de Solaris qui ont été implémentées dans la librariries partagée `libdl.so` par Eric Youngdale et d'autres (voir la référence [30]). La première fonction à utiliser est celle qui permet de charger une librairie partagée:

```
void * dlopen(char * filename, int flag)
```

Le paramètre `filename` contient le chemin et le nom de la librairie partagée à charger. Si ce chemin n'est pas absolu (c.-à-d. il ne commence par /), alors la librairie sera recherchée parmi les chemins `/usr/lib`, puis `/lib` et enfin dans les endroits spécifiés par le fichier `/etc/ld.so.conf`.

Le paramètre `flag` peut être `RTLD_LAZY` indiquant que les symboles non définis de la librairie sont résolus lors de l'exécution de celle-ci ou `RTLD_NOW` indiquant que tous les symboles doivent être résolus avant que la fonction `dlopen` rende la main. Le paramètre `RTLD_GLOBAL` peut être combiné (par un *or*) avec `flag` afin que les symboles chargés par la nouvelle librairie soient disponibles pour les autres librairies chargées par le programme.

La fonction `dlopen` retourne alors une référence (*handle*) vers la librairie chargée. Cette référence sera utilisée plus tard pour accéder aux symboles de la librairie. Si ce *handle* est `NULL`, c'est qu'un problème est survenu durant le chargement. Une description de ce problème peut être obtenue en appelant la fonction `dlerror` dont la signature est

```
char * dlerror(void)
```

Une fois que la librairie est chargée et que l'on dispose donc de son *handle*, il reste à obtenir les pointeurs sur les symboles à utiliser. Ces pointeurs sont obtenus grâce à la fonction `dlsym` dont la signature est donnée ci-dessous:

```
void * dlsym(void * handle, char * symbol)
```

Une fois de plus, si le pointeur renvoyé est `NULL`, c'est que la recherche du symbole a échoué. C'est encore un appel à `dlerror` qui donnera une description du problème rencontré.

Une fois que la librairie n'est plus utilisée, elle peut être détachée du programme par un appel à la fonction `dlclose`:

```
int dlclose(void * handle)
```

Exemple Voyons comment accéder à la fonction `SNVT_set` dont on a parlé au début de cette section:

```
#include <dlfcn.h>
#include <iostream.h>

int main()
{
  void * pHandle;
  void * pSymbol;

  pHandle= dlopen("/home/bquoitin/Tests/lib/libLonTalk.so",
                  RTLD_LAZY);
  if (pHandle != NULL) {
    pSymbol= dlsym(pHandle, "SNVT_set__FPcPci");
    if (pSymbol != NULL)
      ((void(*)(char *, char *, int)) pSymbol)("node_dimmer",
                                               "intensity", 127);
    else
      cout << dlerror() << endl;
    dlclose(pHandle);
  } else
    cout << dlerror() << endl;
}
```

Le nom du symbole passé à la fonction `dlsym` a été concaténé avec sa signature: `SNVT_set__FPcPci`. En effet, il est tout à fait possible que la librairie `libLonTalk.so` contienne plusieurs symboles `SNVT_set` ayant des signatures différentes et il faut que le système puisse les différencier. La signature est donnée par une suite de codes représentant les types des paramètres attendus par la fonction. Dans notre cas, on a d'abord un pointeur sur caractère (`char *`) dont le code est "`Pc`" suivi d'un second pointeur sur caractère et enfin un entier (`int`) représenté par "`i`".

Notez qu'il est possible de connaître ces signatures en utilisant l'utilitaire `nm` qui donne une liste des symboles définis dans une librairie ou un objet binaire. Par exemple,

```
[bquoitin@orion bquoitin]$ nm libLonTalk.so
000006d4 T SNVT_get__FPcPc
00000694 T SNVT_set__FPcPci
00001874 A _DYNAMIC
00001840 A _GLOBAL_OFFSET_TABLE_
00001834 ? __CTOR_END__
00001830 ? __CTOR_LIST__
(...)
```

B.4.2 Threads POSIX

Les threads utilisés par le LonOS sont ceux qui ont été créés par Xavier Leroy de l'INRIA et dénommés LinuxThreads (voir la référence [30]). Il s'agit d'une implémentation partielle des threads POSIX. Celle-ci prend place au sein de la librairie `libpthread.so`.

Les fonctions offertes sont entre autres: la création d'un thread, la définition de ses attributs, l'arrêt d'un thread, l'attente de la terminaison d'un thread.

B.4.3 Sémaphores System V (IPC)

Les sémaphores utilisés sont conformes à la définition System V et sont abondamment décrits dans [30], [16]. Cette implémentation a été préférée à la version POSIX car elle permet de détecter qu'un outil de synchronisation a été supprimé, ce qui n'était pas réalisable avec l'implémentation actuelle de la version POSIX. On pourra également trouver une description des outils de synchronisation tels que les sémaphores et les outils de communication entre processus dans les références [22], [28], [29] et [19].

Liste des acronymes

API	Application Programming Interface
CAL	CAN Application Layer
CAN	Controller Area Network
CMIP	Common Management information Protocol
CMIS	Common Management Information Service
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Code
CSMA	Carrier Sense Multiple Detection
DAB	Distributeur Automatique de Billets
DLL	Dynamic Linked Library
DSAC	Distributed Systems Administration and Control
DSO	Dynamic Shared Object
FIFO	First In First Out
GAB	Guichet Automatique de Banque
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
I^2C	Inter Integrated Circuit
IP	Internet Protocol
ISO	International Organization for Standardization
JDK	Java Development Kit
JNI	Java Native Invokation
LON	LonWorks Operating Network
MAC	Medium Access Control
MIB	Management Information Base
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
RPC	Remote Procedure Call
SMAP	Systems Management Application Process

SNA	Systems Network Architecture
SNMP	Simple Network-Management Protocol
SOW	SML Object Wrapper
TCP	Transmission-Control Protocol
VLSI	Very Large Scale Integration
XIF	eXternal Information File
XML	eXtensible Markup Language

Glossaire

Accesseur: Procédure destinée à accéder à la valeur d'un attribut accédé. On distingue l'accesseur de lecture appelé *get* et l'accesseur d'écriture appelé *set*.

Actuateur: Matériel destiné à modifier un état physique de l'environnement. Il peut s'agir d'un moteur, d'une résistance chauffante, etc.

Attribut accédé: Attribut d'un objet dont la valeur se trouve dans l'entité représentée par l'objet et qui doit être manipulée par des *accesseurs*.

Attribut stocké: Attribut d'un objet dont la valeur se trouve stockée dans l'objet.

Broadcasting: Il s'agit, dans notre cas, d'une transmission vers tous les noeuds connectés à un réseau (diffusion).

Capteur: Matériel destiné à appréhender un état physique de l'environnement. On parlera de capteur de mouvement, de température, etc. On pourra également utiliser le terme *senseur* ou *détecteur* (de fumée, par exemple).

Débitmètre: Le débitmètre est un capteur destiné à mesurer le débit d'un liquide.

Dimmer: Actuateur contrôlant l'intensité lumineuse délivrée par un éclairage.

Electrovanne: Actuateur contrôlant le passage d'un liquide dans un conduit.

Manchester: L'encodage de Manchester désigne une manière de transformer des signaux binaires à transmettre en signaux électriques à émettre sur un support de communication. Le codage de Manchester est le suivant: un bit 1 est transformé en une transition d'un niveau électrique haut vers un niveau électrique bas tandis qu'un bit 0 est transformé en un signal électrique subissant une transition d'un niveau bas vers un niveau haut (voir les références [24], [32]). L'encodage différentiel de Manchester est une variante du précédent présentant une meilleure immunité au bruit.

Firmware: Logiciel destiné au contrôle d'un matériel. Dans ce document, le terme *firmware* peut désigner le logiciel exécuté par les différents processeurs d'un *Neuron Chip* non modifiable par le développeur ou le logiciel mis au point par le développeur et non modifiable par l'utilisateur final du noeud.

Framework: Structure logicielle de communication permettant la transmission de messages entre les entités qui lui sont connectées.

Multicasting: Il s'agit d'une transmission vers un certain nombre de noeuds d'un réseau. Dans le cas du protocole LonTalk (voir le chapitre 1), il s'agit de la transmission vers tous les membres d'un groupe.

Noeud: Unité matérielle de traitement ou point de contrôle connecté à un réseau. Voir noeud LonWorks au chapitre 1.

ObjectClass: Une *ObjectClass* est une classe d'objets dans le modèle objet du LONOS. Elle est donc l'abstraction d'un ensemble d'objets qui ont la même structure et le même comportement. Ces objets représentant des ressources telles que des noeuds, des connexions de variables-réseau (voir la section 1.2). A cette effet, une *ObjectClass* diffère d'une classe d'objets habituelle par le fait qu'elle dispose de variables et de méthodes spécialisées.

ObjectInstance: Une *ObjectInstance* est une instance d'une *ObjectClass*, c.-à-d. qu'elle est un objet appartenant à l'ensemble abstrait par l'*ObjectClass*. Le fait qu'une *ObjectInstance* soit une instance d'une *ObjectClass* indique qu'elle représenté une entité réelle ou virtuelle dont le type correspond à cette *ObjectClass*.

PPP-CSMA: Il s'agit d'une méthode d'accès à un support de communication partagé entre plusieurs noeuds. Rappelons d'abord que CSMA signifie Carrier Sense Multiple Access, c.-à-d. qu'un noeud est capable de détecter si le support est libre. Le "Predictive p-persistent CSMA" est basé sur une méthode statistique qui consiste à garder une évaluation de la charge du support pour déterminer quand y accéder. Ce protocole essaye donc d'éviter les collisions avant tout.

Transceiver: Matériel permettant l'encodage d'informations binaires en signal électrique. Cet encodage est encore appelé signalisation.

Unicasting: Par opposition à broadcasting et à multicasting, il s'agit d'une transmission vers un noeud.

Bibliographie

[1] *Connecting LonWorks and TCP/IP Enterprise Networks - Real Application Successes*, Coactive Aesthetics, Inc. 1997.

[2] *Facility Executives' Guide to Interoperability - Making the right decisions today can assure that buildings remain efficient and flexible well into the 21st century*, Alan Pearlman, Tom Grimard, Electronic Systems Associates, 1997.

[3] *LonWorks Technology Devices: MC143150, MC143120*, Motorola Inc. 199x.

[4] *LonTalk Protocol Specification, version 3.0*, Echelon Corporation, 1994.

[5] *LonWorks Engineering Bulletin: LonTalk Protocol*, Echelon Corporation, 1993.

[6] *SNMP, SNMP v2 and CMIP - The Practical Guide To Network Management Standards*, William Stallings, Addison-Wesley, 1993. ISBN: 0-201-63331-0.

[7] *The Common Object Request Broker - Architecture and Specification*, OMG, v2.2, 2/1998.

[8] *The omniORB version 2.6 - User's Guide*, Sai-Lai Lo, Olivetti & Oracle Research Laboratory, 30/9/1998.

[9] *CORBA Async Requirements*, Ron I. Resnick, 1996.

[10] *Java Native Interface Specification - release 1.1*, JavaSoft - Sun Microsystems Inc. Business, May 1997.

[11] *Java Beans Specification*, JavaSoft - Sun Microsystems Inc. Business.

[12] *Java Object Serialization Specification*, JavaSoft, Sun Microsystems Inc. Business, october 8, 1997.

[13] *Software Development for Intranet Applications*, Dan Marinescu, Computer Sciences Department, Purdue University.

[14] *A Gentle Introduction to BOND*, Computer Sciences Department, Purdue University, November 2, 1998.

[15] *BOND Objects - a white paper*, Ladislau Boloni, Computer Sciences Department, Purdue University, February 19, 1998.

[16] *La communication sous UNIX — applications distribuées, 2ème édition*, Jean-Marie Rifflet, Ediscience, 1994. ISBN: 2-84074-106-7.

[17] *Linux Device Drivers*, Alessandro Rubini, O'Reilly, 1998. ISBN: 1-56592-292-1.

[18] *C++ Primer, 2nd edition*, Stanley B. Lippman, AT&T Laboratories, Addison-Wesley, 1995. ISBN: 0-201-54848-8.

[19] *Operating System Concepts, 5^{th} edition*, Abraham Silberschatz and Peter Baen Galvin, Addison-Wesley, 1998. ISBN: 0-201-54262-5.

[20] *ISM Framework/CMIS Dispatcher Design*, BULL France, division BullSoft — OpenMaster.

[21] *Cours de Systèmes d'Exploitation*, Jean Ramaekers, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, 1996.

[22] *Cours de Systèmes d'Exploitation - matières approfondies*, Jean Ramaekers, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, 1997.

[23] *Cours de Systèmes d'Exploitation - systèmes distribués*, Jean Ramaekers, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, 1997.

[24] *Cours de Téléinformatique et Réseaux*, Philippe van Bastelaer, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, 1997.

[25] *Cours de Téléinformatique et Réseaux - matières approfondies*, Philippe van Bastelaer, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix,

1997.

[26] *LNS Host API Guide*, Echelon Corporation, 1994-1997.

[27] *Cours de sécurité*, Jean Ramaekers, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, 1998.

[28] *The Linux Kernel*, David Rusling, The Linux Documentation Project, 1996-1999.

[29] *The Linux Programming Guide*, Sven Goldt, Sven van der Meer, Scott Burket, Matt Welsh, The Linux Documentation Project, 1995.

[30] *"Man pages" de dlopen, dlsym, dlclose, pthread_create, pthread_cancel, semctl, semget, semop, ...*, Linux RedHat 5.2, 1998.

[31] *Essential Client/Server Survival Guide — 2nd edition*, Robert Orfali, Dan Harkey & Jeri Edwards, Wiley, 1998. Traduction de François Leroy et Jean-Pierre Gout, Vuibert. ISBN: 2-7117-8624-2.

[32] *Réseaux — 3ème édition*, Andrew Tanenbaum, InterEditions, 1997. ISBN: 2-7296-0643-2.